THE COMPLETE RAINBOW GUIDE TO OS-9

By Dale L. Puckett and Peter Dibble

From the publishers of
THE RAINBOW,® The Color Computer Monthly Magazine

THE COMPLETE RAINBOW GUIDE TO OS-9

By Dale L. Puckett and Peter Dibble

Falsoft, Inc.
Prospect, Kentucky

THE COMPLETE RAINBOW GUIDE TO OS-9

Editor: Courtney Noe
Editorial Assistant: Tamara Solley
Illustrations and Cover Design by Jerry McKiernan

The Rainbow Bookshelf™ books are published by Falsoft, Inc., Lawrence C. Falk, President.

First published in 1985.

ISBN: 0-932471-00-5
Library of Congress Catalog Card Number: 85-70113

Printed in the United States of America
1 2 3 4 5 6 7 8 9 10

table of contents

PART II—HANDS ON

PART III—TOURING THE OS-9 COMMAND SET

PART IV—PROGRAMMING LANGUAGES

PART V—TOWARD THE END OF THE RAINBOW

The Complete Rainbow Guide To OS-9 is just that, the most comprehensive tutorial and reference guide yet published for the multifaceted OS-9 operating system.

The book restates Falsoft's commitment to a series of books designed to broaden the base of knowledge about the Color Computer and, in the process, to help us to realize the vast potential of this incredible tool.

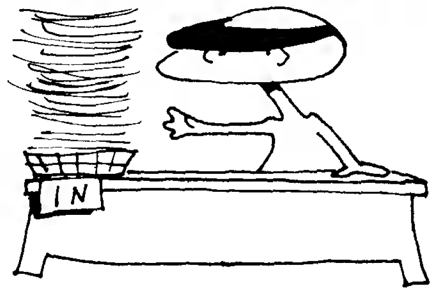
The credit for this extraordinary guide goes to Dale Puckett and Peter Dibble, who, as two of the foremost authorities on OS-9, have taken great pains to ensure that the book is of immediate value to the average CoCo user. I think you will appreciate their writing style and will agree that this book is a milestone in the evolution and utility of the OS-9 system. I also should thank Ken Kaplan, president of the Microware Systems Corporation, for his encouragement and assistance in the preparation of the book, as well as for adapting OS-9 for use with the Color Computer.

You will especially enjoy the many sample programs that the authors have included to ease the learning process for you. These programs, you'll be glad to know, are available on The Rainbow Guide To OS-9 Disks if you want to save yourself hours of time typing in the listings (see the notice elsewhere in this book).

I hope you enjoy The Complete Rainbow Guide To OS-9 as much as we delight in making it possible.

Lawrence C. Falk
Publisher

why we wrote this book



Since its release on the Radio Shack Color Computer in October 1983, Microware's OS-9 Operating System has created a stir. Power-packed and efficient, OS-9 brought a UNIX-like environment to an inexpensive microcomputer for the first time.

Oldtimers and hackers revelled in its power. Many beginners however, found it intolerable.

After answering hundreds of questions in "KISSable OS-9," a monthly column published in THE RAINBOW, we discovered a pattern. People with no computer training or experience were rushing to their local Radio Shack and buying OS-9. Then, they rushed home, proudly inserted their new operating system and went to work.

With little fanfare and without too much difficulty, these converts learned to build files and list them to their CoCo's screen. Some even learned how to climb around on OS-9's directory tree. But eventually the honeymoon ended, and a lot of new OS-9 users discovered that they didn't have the slightest idea about what to do with their new operating system.

Experienced users who had learned how to program using the Color Computer's Microsoft BASIC knew what they wanted to do. But OS-9 proved an alien environment to many. They found themselves lost in a reference manual that gave experienced mainframe programmers everything they needed, but left beginners wondering where to start.

Enter "The Complete Rainbow Guide To OS-9" from THE RAINBOW. Here, we'll try to lay down a foundation that will let you build a staple of OS-9 programming skills with ease.

USING THIS BOOK

We've divided "The Complete Rainbow Guide To OS-9" into six parts.

Part I presents an overview of OS-9. It gives you "The Big Picture." We encourage you to leave your computer off as you study this section.

In Part II we hope you will turn your computer on, use our examples and experiment. This is where you get your "Hands On" OS-9.

Part III takes you on a seven chapter tour of the complete OS-9 Utility Command Set. Each chapter introduces you to a number of commands that perform logically related functions.

In Part IV we introduce you to the major programming languages that run on OS-9 computers. You'll also be introduced to assembly language programming and the "toolkit" concept here.

Part V moves you "Toward the End of The Rainbow." Here you'll be able to look inside OS-9 and explore its inner workings. You'll be in hacker heaven.

And finally, in Part VI we'll show you the "Pot of Gold" that lies at the end of the rainbow. Here we dig into the internal workings of OS-9 and list a new device driver that's worth its weight in gold. The additional sample modules are a bonus.

CHAPTER DESCRIPTIONS

In Chapter 1, we'll look at OS-9's history. We'll also try to answer a few of your more obvious questions. What is an operating system? Why do I need one? What is UNIX? Who designed it? What is OS-9? What will it do for me?

Chapter 2 takes a look at OS-9 Hardware. It describes the minimum requirements for OS-9 based systems and looks at two important system parts — Device Descriptors and Device Drivers. These two software modules and the new hardware are all that you need to expand an OS-9 system.

In Chapter 3 we show you how OS-9 uses memory. You'll learn that the system is divided into program modules and data modules. We also show you how OS-9 manages your computer's memory.

Chapter 4 describes OS-9 Software. You'll learn that OS-9 has four major parts. The "Kernal" manages your computer's memory and your microprocessor's time. It gives OS-9 the ability to do more than one thing at a time and let's you use more than one

terminal at the same time. Surrounding the Kernal is OS-9's "Shell." It translates your commands into a form your computer can understand and passes them to the Kernal for action. Just outside the Shell, OS-9's "Utility Command Set" gives you the tools you need to do many routine jobs. And finally, "Application Programs" are tools you use each day at work or at play.

We describe OS-9's file system in Chapter 5. Here you'll learn about execution directories, data directories and anonymous directories. We'll show you how to list these directories and how to delete them. "Files" live in directories and in this chapter we'll show you how to find them, use them, protect them and share them. We'll even convince you that a device like your printer is also a file.

Chapter 6 shows you how OS-9 communicates with the outside world. You'll learn about standard input, standard output and find out that you can redirect OS-9's input and output to files and other hardware devices. We'll also introduce you to "pipes" and the concept of filters. You'll discover that you can run several small programs at the same time to do one big job.

In Chapter 7 we introduce you to the OS-9 System Disk as we introduce you to The OS9Boot File, the SYS, CMDS and DEFS directories and show you how to use OS-9's "startup" file. After these introductions, we'll show you how to "boot" your system, format new disks, make an exact backup of your original system disk and how to copy everything on it to another disk with a different format.

Chapter 8 introduces you to several special keys that will let you streamline your OS-9 operations and minimize your keystrokes. We'll also show you which keys to strike on your Color Computer keyboard when you need to generate special characters like the left and right brackets required by many of the advanced programming languages that run under OS-9.

In Chapter 9 you'll be introduced to a few basic operating system functions. You'll learn how to BUILD files, LIST files, and how to use them. We'll even show you how to RENAME and DELete them.

Chapter 10 demonstrates commands that give you information. In Chapter 11 you'll learn about commands that do things to files. We introduce commands that work with directories in Chapter 12, and show you the commands you need when you want to create and copy your system in Chapter 13. Chapter 14 deals with commands that act on the system, and Chapter 13 talks about a group of special commands that are always in memory waiting for you to use them.

In Chapter 16 we try to bring everything together as we introduce you to procedure files. We'll show you procedures that will let

you change the boot file on your system disk and change the stepping rate of your disk drives. We'll even document a few known bugs in CoCo OS-9, and where possible, give you the fixes. Finally, we'll show you how to make changes in memory from your "startup" file.

In Chapter 17 we'll introduce you to the concept of using small programming "tools." You'll learn about several of the "toolkits" that are available commercially, and we'll show you what they can do for you and how they differ.

Chapter 18 introduces you to assembler language and shows you how to get started with the OS-9 assembler, ASM.

"High Level Languages" come to center stage in Chapter 19 as we present a brief overview of BASIC09, C, and Pascal. We'll show you the difference between interactive languages like BASIC09 and batch oriented languages such as C. There won't be a lot of discussion here — each language is a book in itself — but we will present a few programs that show OS-9 programming languages in action.

Chapter 20 shows you how to manage your memory. You'll learn how to conserve this precious resource and how to deal with fragmentation. Along the way, you'll learn what makes OS-9 modules reentrant.

Chapter 21 is to disk space what Chapter 20 is to memory. We'll teach you to avoid pitfalls like small files and fragmented files. Then, we'll show you how to recover files if you delete them by mistake or damage your disk.

Chapter 22 is about device descriptors. You'll learn several ways to create new device descriptors to modify the characteristics of devices you have or accommodate new hardware you may get.

Chapter 23 explains the role of device drivers in the OS-9 system. You'll learn where they fit in and what's involved in writing your own. You'll come away with a list of reasons for tackling the job. The details of driver writing are left for you to learn from examples in the Workshop section.

Chapter 24 jumps sideways into the area of processes. Processes are the muscles of an OS-9 system. We'll teach you several ways to use processes and how to control them. Look for information about tuning your system for maximum performance and using signals. This chapter includes a sample program that is pure mischief.

Chapter 25 moves back into the I/O path. You'll discover how I/O managers fit into the scheme of things. Each of the three main file managers is discussed, and we speculate about a number of file managers that aren't available yet.

Chapter 26 moves up the I/O ladder to the executive position. We'll show you why IOMan deserves the memory it uses.

Chapter 27 is an overview of the OS-9 disk format. First, you'll learn why a disk needs to be formatted. You'll also find out about the low-level structure of an OS-9 disk. We don't go as low as the physical structure of the disk, but if you can read something on a disk from a program, you'll learn about it in this chapter.

Chapter 28 covers interrupts, the heartbeat of every OS-9 system. We'll teach you what interrupts are, why they're important and what OS-9 does about them.

Chapter 29 goes very deeply into modules. Some of the material in this chapter has been covered earlier in the book, but this chapter takes it further than any but the true hacker will want to go.

Chapter 30 covers memory the way Chapter 29 covers modules. It starts with an overview of the theory behind memory management, including a careful explanation of fragmentation. You'll learn about the way OS-9 handles memory internally. At the end of the chapter, you'll find a discussion of each system service request that relates to memory management under OS-9 Level One.

Chapter 31 continues the subject of memory management. This chapter homes in on memory management under OS-9 Level Two.

Chapter 32 will teach Level Two programmers how to use the DAT. We'll teach you about each system service request that is involved with Level Two memory management.

THE WORKSHOP

The workshop is a selection of programs. Most of these programs should prove useful, but mainly they are examples. By referring to the workshop, you can find out how to use shared memory modules, how to send and receive signals, how to fork a process, how to write a device driver and lots of other things.

Some of the programs are: a Daemon, four different device drivers (two directly from Microware), a program that controls output to your terminal, and a program that runs other programs at special priorities.

MEMORY MAP

These memory maps are pictures that show the way OS-9 control blocks are hooked together. If you need to find the value of some system variable, you'll find these maps useful. There are two maps; one for Level One, the other for Level Two.

A warning: these maps are for OS-9 version 1.2. They may not work for any other version.

HOW THIS BOOK WAS CREATED

The manuscript of this book was prepared by the authors using the DynaStar word processing program. Spelling accuracy was checked with the DynaSpell spelling checker. Procedures listed throughout the book were run on a 6809-based GIMIX microcomputer running the OS-9 Level Two operating system, and a Radio Shack Color Computer running OS-9 Level One. They were then copied directly into the manuscript.

The authors shipped the manuscript to the publisher on a standard OS-9 disk.

ABOUT THE AUTHORS

Dale L. Puckett is a freelance writer and programmer who first learned about bits, bytes and BASIC when he built his first "television typewriter" — a SWTPC CT-1024 — in 1975. When the keyboard didn't arrive with his kit, he wired a set of nine slide switches together and put his first message on the screen one byte at a time.

A month later, he built a SWTPC 6800 microprocessor with 12K of memory and has been programming every since. A cassette storage unit wasn't available then, so he often left his computer on for weeks at a time after finishing a long program.

His programs — sold by the Frank Hogg Laboratory — include DynaSpell, Esther, Help, Lk and Readtest. He also designed and is co-author of "The Speller," which runs on the IBM PC and Apple computers.

Dale is presently a contributing editor to THE RAINBOW and author of the "KISSable OS-9" column in that magazine. He also serves as the President of the OS-9 Users Group, an Iowa Corporation with members worldwide. He has served on the InfoWorld review board and has written for Hot CoCo, '68 Micro Journal and Micro.

An amateur radio operator, K0HYD, since 1956, he has held a first class radiotelephone operators license since 1962. He has worked at several radio and television stations in Kansas and New Jersey.

Dale is a chief warrant officer in the United States Coast Guard and presently serves with the Pollution Response Branch at Coast Guard Headquarters in Washington. He lives in Dale City, VA with his wife Esther and daughter Michele.

Puckett received a Bachelor of Science degree from the William Allen White School of Journalism at the University of Kansas

in 1966. He also earned a Master of Arts in Management from Webster College at St. Louis, Mo.

Peter Dibble, born in Waterbury, Connecticut, received the degree of Bachelor of Science in chemistry from the University of Connecticut. Subsequent to graduation, he has held jobs as an application programmer, a systems programmer and the assistant director in charge of the University of Rochester Computing Center's user services department. He is now a graduate student in the University of Rochester computer science department. He has been writing a monthly column called "OS-9 User Notes" for *'68' Micro Journal* since April 1983.

ACKNOWLEDGEMENTS

We thank Lonnie Falk, our publisher; Courtney Noe, our editor; Tamara Solley, editorial assistant and typesetter; Jerry McKiernan, our artist; and the entire staff at THE RAINBOW. Without their encouragement and support, this book would have never been published. They demonstrated faith to let us try and the patience to let it work.

My wife, Esther Puckett, who patiently watched while I searched for words that wouldn't come, and who edited those that didn't work, deserves much of the credit for the success of this book.

Special thanks go to two OS-9 programmers who contributed programs published in this book. Tim Harris, a student at the University of Iowa at Ames, is the author of several assembly language and C filters that appear in Chapters 18 and 19. Bill Ball, a U.S. Coast Guard public affairs specialist assigned to the Defense Information School at Fort Benjamin Harrison, Indiana, contributed a file report utility written in C. Named Frep, it is presented in Chapter 19.

CHAPTER 1

the historical connection

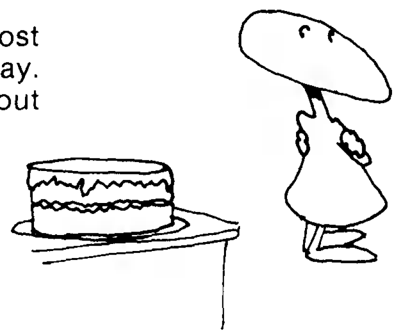
"I will liken him unto a wise man, which built his house upon a rock: and the rain descended, and the floods came, and the winds blew, and beat upon that house; and it fell not: for it was founded upon a rock." (Matthew 7:24-27)

COMMON SENSE PREVAILS

Temptation threatened. We wanted to let you put your fingers on the keyboard and work with OS-9 immediately. In fact, we could hardly wait to show off our favorite operating system.

Common sense prevailed, however, and we decided to present an expanded introduction first. This approach will give you a strong foundation upon which you can build a staple of OS-9 programming skills.

An introduction to OS-9 is an introduction to one of the most versatile operating systems available on a microcomputer today. But before we introduce you to OS-9, we need to talk briefly about operating systems.



In this chapter you'll be introduced to:

Operating Systems
UNIX History
OS-9 History

OPERATING SYSTEMS TALK TO YOUR HARDWARE

Let's jump right in with the sixty-four dollar question. What is an operating system?

First, we'll give a formal answer. An operating system controls the low-level processes within your computer. It gives your appli-

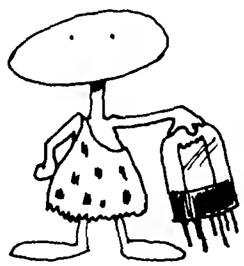
cation programs a way to communicate with the hardware. It also manages your system resources.

Processes are short software routines in action. They get characters from a keyboard. They put characters in a disk file or send them to a printer. Sometimes, they even make you think they are doing several things at the same time.

Externally, system resources include terminals, printers, plotters and disk drives, etc. Internally, they include your computer's memory and your microprocessor's time.

Enough formality. An operating system directs the flow of your data. Like a traffic cop on a busy corner, it insures that the right data gets to the right place — at the right time.

PEOPLE WERE THE FIRST OPERATING SYSTEMS



In the early days, when computers with far less power than your lap computer filled large air-conditioned rooms, men and women stood watch. They threw hundreds of switches to enter a short program. They received reports on slow teletype printers. They even watched the vacuum tubes and replaced them when they burned out. They were the operating systems.

Today, as system software developers struggle to keep pace with the ever increasing demand for microcomputer power on American desks, operating systems are becoming total environments.

Keyboards, video displays, floppy disk drives and printers are child's play to today's operating systems. Application programs now demand graphics, light pens, touch screens and windows. Users demand mice drawing tablets and other "friendly" tools.

Now that you know what operating systems are supposed to do, you can compare them to a piece of software you are already familiar with — MicroSoft's Extended Color BASIC.

Color Disk Extended BASIC is a programming language. It is called "Disk" Extended BASIC because it contains a few simple routines that let you save programs and data on a floppy disk and load them back in later.

By contrast, OS-9 is an operating system — an environment. It connects your program to the keyboard you type on. It writes letters and numbers on your video screen. It sends listings to your printer. It saves your programs on floppy disks. It even lets you run two or more programs at the same time.

OS-9 is, in fact, an extremely efficient implementation of the UNIX operating system philosophy. It was designed by Microware Systems Corporation in Des Moines, Iowa. And because it was coded in 6809 assembly language, it is small and fast.

OS-9 HAS ROOTS IN UNIX

Long ago, about 1969, and far, far away — in a small, northern New Jersey hamlet — AT&T designers were forced to deal with reality. Software development was too expensive.

Managers visited. They saw expensive computers tied up with only one person doing one job. It didn't make sense to them. They wanted to make more money. They wanted efficiency. UNIX was born.

In those days, programmers had to use batch-oriented operating systems. To get programs or data into a computer, they typed it onto coded punch cards. Several minutes — and many times, hours — later, they received a printout with their results. The system was too slow. They needed an interactive system.

For a short while in 1969 they used an operating system called "Multics." This new program let several programmers work at the same time. It even let them use more than one program at the same time.

Multics also was interactive. This meant that when programmers typed a command, the computer responded almost immediately. It was a big improvement over the punch cards.

But Multics wasn't enough. Programmers needed an operating system that would support coordinated teams working on the same product. They needed to be able to access each others data. In fact, the name UNIX came from a reference to this unified, team programming environment.

The first UNIX system was developed by a Bell Laboratories scientist named Ken Thompson. He wrote it in assembly language and ran it on a PDP-7 minicomputer.

UNIX IS WRITTEN IN C

There was one serious problem with Thompson's approach. Since his operating system was written in assembly language, it was machine dependent. It could only be run on one model or type of computer. If you needed to run it on another computer, you had to code it over again in the assembly language recognized by the new computer. Again this was too expensive.

To solve this problem, Thompson wrote a language that could be transported to other computers. He named it B. Later, another Bell programmer, Dennis Ritchie, modified Thompson's B and called the new language C. C is still used today by programmers writing system utilities and application programs for almost every computer on the market.

After developing C, Ritchie rewrote UNIX in C. Since C could

be moved to virtually any computer, programmers at Bell Labs could put UNIX on just about any machine they wanted. In fact, by 1978 they had installed more than 600 UNIX systems at universities, government facilities and throughout the Bell System. These machines controlled laboratory experiments, designed machines, supervised telephone networks and managed business offices.

In 1979, Bell Laboratories introduced a new version of UNIX and decreased its single-user license fee. Computer manufacturers could then afford to develop UNIX software for their systems, and small businesses could license UNIX for their personal computers. System III UNIX was released in 1981. Now, System V is on the market.

The UNIX operating system contains three major parts: a kernal that schedules tasks and manages data storage, a shell that interprets the commands typed by the user, and an extremely large set of utility programs that perform hundreds of routine tasks and system maintenance.

Since it was designed by programmers *for* programmers, it contains a wealth of program development tools. With UNIX, you can do many jobs without a programming language. You simply place a sequence of system commands in a shellscript.

OS-9 BRINGS UNIX POWER TO THE 6809

UNIX was the answer to a programmers prayer. But, since it was written in C, it was too big to fit comfortably on most microcomputers. Yet, many companies saw the need for UNIX power on these small machines.

Why did they perceive a need? Stop for a moment and think about the work you do with your computer.

How many times have you wanted to use one program while you were running another? How many times have you wished that your spouse could use the computer to keep your household records straight while you were programming it from a second terminal? Have you ever wished you could print a long report and compose another at the same time?

Both Microware Systems Corporation and Motorola saw the need clearly. They joined forces to make it happen for the 6809 microprocessor.

Motorola layed down some tough criteria. They wanted an operating system that would exercise every ounce of capability in the 6809. Several 16-bit registers and almost every memory addressing mode available on a minicomputer made the job easier.

The company's ultimate goal was to sell mass-produced

"software-on-silcon." Motorola wanted to distribute their software products in ROM (Read Only Memory) chips. This meant that they had to be able to write software in small modules that could be plugged in anywhere in memory.

The use of an assembler to reassemble the source code, or a linking loader to link modules together at "run time," was out of the question. These techniques create too much of a hassle for the ultimate consumer. To Microware, all of this meant the new operating system had to be written using "position independent" modules.

The new operating system also had to meet several additional requirements. All modules had to be reentrant because more than one user would be running them at the same time. Programmers had to be able to interrupt a routine while it was running, let another user execute that same routine, and then have it return to the original user with all original data intact. It also meant that programs in the modules could not modify themselves while they were running.

Because of the tough criteria and effective design, software developers have been able to transport just about every major language and all types of system software to OS-9 computers. In fact, most of these programs run much faster on the 6809. And, they're much shorter. The applications software you need to run your business is available, too.

YOU'RE IN GOOD COMPANY WITH OS-9

OS-9 has put the UNIX philosophy to work on Radio Shack's Color Computer and nearly 100 other microcomputers that use the 6809 microprocessor. In fact, several major American companies use OS-9 daily.

For example, the Western Electric division of American Telephone and Telegraph — the same AT&T that hopes to make UNIX a household word — uses a program written in BASIC09, running under the OS-9 operating system on a Gimix microcomputer, in the final manufacturing stage of every telephone that leaves its factory in West Virginia.

OS-9 even helps keep the Space Shuttle flying. NASA uses four Gimix systems running OS-9 at Cape Canaveral, and one of them performs the pre-flight fuel tank tests before every launch.

The Ford Motor Company uses microcomputers running OS-9 at their test track in Michigan. And, Eastman Kodak uses a BASIC09 program during the final assembly of each Kodak disk camera. They use another OS-9 program to ensure the quality of the film manufactured for those cameras.

OS-9 FUTURE IS PROMISING

The demonstrated popularity of OS-9 gives you a good reason to learn this operating system and become proficient in its operation. If you learn to operate and program in the OS-9 environment, you could have a bright future. Additionally, the fact that AT&T hopes to make UNIX *the* standard operating system and the fact that learning OS-9 is an inexpensive way to learn UNIX principles give you other good incentives.

OS-9 is also a tool you can use to keep your applications programs compatible with new 6809 hardware. If you write all of your programs using relocatable code, and use only standard OS-9 operating system calls, you will almost guarantee that your software will run on new 6809 computers. This is especially important if you are writing software for machines like Tandy's Color Computer.

Tandy continues to improve and update its hardware. Yet, company officials know that it is in their best interest to ensure that the old software will run on the new hardware. Because of this, they recently recommended that developers use OS-9. If you are developing software, even for your own use, it is also in your best interest to use OS-9.

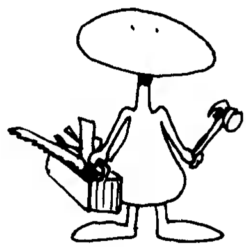
SUMMARY

In this chapter we have reviewed OS-9's historical connection. You have been introduced to operating systems in general and Bell Laboratories UNIX operating system in particular. You have also learned how OS-9 was born.

In Chapter Two we'll look at "The Hardware Connection" and introduce you to a few OS-9 system requirements.

CHAPTER 2

the hardware connection



In the beginning there was a Central Processing Unit. Or, did algorithms come first? Without leaving time to open that debate, we move on to take a look at hardware and the part it plays in computing.

Then, we talk about hardware requirements for OS-9 based microcomputers and introduce you to device descriptors and device drivers — two OS-9 software modules that make it easy to expand your system.

In this chapter you'll learn about:

Central Processing Units
Memory
Data Storage
Input and Output devices
OS-9 Hardware Requirements
OS-9 Device Drivers
OS-9 Device Descriptors

CENTRAL PROCESSING UNITS

A Central Processing Unit in a computer is like the brain in the human body. It interprets and executes each instruction in a computer program.

In the early days, engineers built CPUs with vacuum tubes. They were bulky, hot and slow. But, they got the job done. Later,

transistors replaced vacuum tubes and the newer models became smaller, cooler and faster.

The first microprocessor was introduced in 1972. By the end of that decade small microcomputers had invaded America. Business executives found computing power on their desks that rivaled that of the large mainframe computers of the sixties.

The 6809 microprocessor from Motorola is the CPU that runs OS-9. This single integrated circuit — or chip as it is called — can interpret and execute millions of instructions each second.

MEMORY

The 6809 microprocessor has tremendous power. Yet, it can only deal with a finite amount of data at any given time. A computer must be able to remember — or store — the long data sequences that make up programs if it is to do any useful work. Small integrated circuits handle this chore.

Logically enough, engineers call the integrated circuits that hold this data memory. These chips store information by remembering whether an electronic pulse was on or off. If a positive voltage can be measured at a particular point — or address — that data "bit" is said to be on. A point that measures zero volts is off.

Generally speaking, eight of these bits make up one byte of data. Eight bits of data can hold 256 different patterns. The pattern determines the value of the byte. Alas, each byte of data can have one of 256 different values.

Main memory inside a computer takes one of two forms. Random Access Memory (RAM) can be read from and written to. It is used to hold data that changes during the execution of a program. Read Only Memory (ROM), can only be read. Engineers use it to hold programs or data that never change.

Since OS-9 is a multi-user system, memory is shared by several users. In fact, memory management is one of the main tasks performed by OS-9.

DATA STORAGE

A microprocessor can only address a finite amount of memory at a given time. Because of this, microcomputers need a place to store information that is not being used at the time.

Smaller microcomputers use floppy disks to store information on a flat magnetic surface. They save this data by sending magnetic pulses to the surface of the disk. Later they read the information back using a magnetic read/write head much like those found on home tape recorders. Microcomputers use a disk controller to

move this head to different positions on the disk, allowing access to information stored anywhere on its surface.

Hard disks are also used to store data. Because they use a rigid aluminum plate coated with a magnetic emulsion, they can hold more data and operate much faster. Their increased capacity and speed is needed for most business applications.

Both floppy and hard disks come in various shapes and sizes. Drives used on microprocessors range from the older 8-inch models to the newer 3½-inch compacts. The standard 5 ¼-inch drives, both full and half size, are probably the most common. You can find them all on OS-9 based microcomputers.

INPUT AND OUTPUT DEVICES

Before a CPU can process any information, it must be entered into the computer. Keyboards with a standard typewriter layout are the most common input device today, but engineers have pushed technology so hard during the past several years that mice, touch sensitive screens and voice recognition hardware are rapidly becoming the state of the art.

Most people use a terminal to communicate with their computer. A terminal has a keyboard that lets them send data to the computer, a video display screen that lets them see the data it sends back, and some type of communications port that connects the two. Some terminals even display graphic images.

When you run OS-9 on the Radio Shack Color Computer, the Color Computer acts as both your computer and your terminal. You can also hook a full-sized terminal up to the RS-232 jack on the rear of the Color Computer.

OS-9 HARDWARE REQUIREMENTS

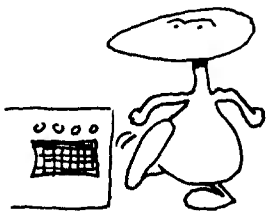
Some OS-9 computers only use one 4K ROM and 2K of RAM. In fact, most engineers don't even call these small machines computers. They call them controllers.

On the other end of the spectrum, others use OS-9 computers that need a million bytes of memory and a large hard disk. But, these giant machines pay for themselves. For example, many schools connect one large OS-9 based computer to a dozen terminals and let 12 students all work at the same time.

If you plan to write all of your programs in assembly language, you can get by with as little as 24K of RAM. If you use a higher level language like BASIC09 or C, you'll need at least 40K of memory.

LEVEL I SYSTEMS

Because memory space is limited, OS-9 Level I is usually only used on computers designed for one person. Most of these machines use 4K of ROM and up to 60K of RAM. The ROM is usually addressed at $F800 and normally runs through $FFFF. The 6809 microprocessor automatically looks for start up vectors at the top of memory. The ROM holds these vectors.



Radio Shack emulates the ROM found in other computers by loading the upper 32K page of RAM in a 64K Color Computer with OS-9 start up code stored on track 34 of a standard Extended BASIC disk. Once the code is in place, the Color Computer's SAM chip turns off the BASIC ROM memory and turns the RAM on. The 6809 then uses the code stored at $F800 to $FFFF to "boot" the rest of the system from a standard Radio Shack OS-9 disk.

Most Level I systems use a terminal that talks with OS-9 through a serial or parallel port. However, some computer manufacturers use a memory mapped video display. The Color Computer uses this approach.

Level I OS-9 can send data to your printer if you own one. And if your computer has a real-time clock chip installed, it will use it. Printers may be either serial or parallel. The Color Computer uses a serial printer, but it does not have a hardware clock chip.

LEVEL II SYSTEMS

OS-9 Level II systems use memory management hardware—a chip called a DAT. The letters DAT stand for dynamic address translator. The DAT chip makes it possible for the 6809 microprocessor to use more than 64K of memory. Normally, the 6809 cannot address more than 64K of memory. However, the DAT chip moves memory in and out of a fixed 64K block that the 6809 addresses.

An OS-9 Level II computer needs 64K of RAM for the operating system, plus 32K of RAM for each user. It also needs a main terminal and a terminal for each person who is going to use it. OS-9 Level II computers store their data and programs on both floppy and hard disks.

DEVICE DRIVERS

Engineers use computers running OS-9 to control all types of hardware. They prefer OS-9 because they can connect new hardware tools to their computer without rewriting or patching the operating system.

OS-9 lets you add new hardware to your computer by adding two new software modules — a device driver and a device descrip-

tor. After the new hardware is connected to your computer, you load your two new modules into memory. Once these modules are in place, you can receive data from the new device or send data to it by redirecting your standard input and output paths to the new hardware.

For example, you could build a voice synthesizer and plug it into one of the expansion slots in the Radio Shack Multi-Pak Interface. To run it, you would write a device driver — you could call it voice — and a device descriptor named ' V '. To make your voice synthesizer read your directory you would type this command line:

OS9: dir >/V <ENTER>

Device drivers are short pieces of 6809 code that are smart enough to know how to talk to a particular piece of hardware. Most OS-9 computer systems come with device drivers that know how to talk to the Asynchronous Communication Interface Adapters or ACIAs that connect them to serial terminals and printers as well as to the Peripheral Interface Adapters or PIAs that let them communicate with parallel keyboards and printers. They also come with special drivers that know how to communicate with the particular disk controller used by each manufacturer.

When OS-9 programmers write a device driver, they use reentrant code. Because the driver is reentrant, only one copy needs to be in memory — even though several pieces of hardware may be using it at the same time.

For example, ACIA — the standard OS-9 device driver — often appears to be talking to your terminal and sending a listing to your serial printer at the same time. Chapter 24 explains device drivers in great detail and shows you how to write one.

DEVICE DESCRIPTORS

An OS-9 device driver uses a device descriptor when it talks to a piece of hardware. The driver itself is generic. This means it can talk to any piece of hardware that uses the same chip. For example, ACIA, the device driver we mentioned earlier, can talk to any device that uses an ACIA. It can send to and receive characters from a terminal. It can send information into a modem connected to a telephone which lets you communicate with a remote computer. And, it lets you send listings to your printer.

To talk to the terminal, OS-9 uses a device descriptor named /TERM. When you are working with OS-9 it normally gets the characters you type on your keyboard from this device descriptor. Likewise, the characters you see on your screen come through it.

To feed the modem, you would redirect your standard output path to a device descriptor named /M. To list a file to the printer,

you would redirect your output to a device named /P. The two command lines used to feed your modem or printer would look like this.

OS9: list Chapter_One >/M <ENTER>
OS9: list Chapter_Two >/P <ENTER>

These device descriptors do what their name implies. They describe the actual physical characteristics of the hardware for the device driver. Without them, the device driver would be lost. The device driver needs to know the name of the device it is supposed to send data to, its location — or address — in memory, and the name of the file manager that will be sending the the data. It gets all of this information from the device descriptor.

A device descriptor is nothing more than a small table that holds information that describes each piece of hardware. It also holds the name of the file manager that will be sending data to the device, the name of the driver that will use it, the absolute physical address of the device in your computer's memory map and an initialization table.

The initialization table contains information that describes the initial characteristics of a piece of hardware. For example, it may tell OS-9 to send an ASCII backspace character, 8 decimal, when it wants to back the cursor up one space.

Nearly two dozen characteristics — called parameters — are stored in a typical OS-9 initialization table. They can be read or changed with the XMODE utility command. In Chapter 23 we describe device descriptors line by line and show you how you can build them.

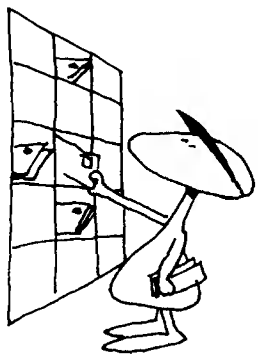
SUMMARY

In this chapter you have learned about computer hardware in general and minimum hardware requirements for the OS-9 operating system. You have been introduced to device drivers and device descriptors, two software modules that make it easy for you to install new hardware on your computer.

In Chapter 3 we introduce you to the concept of modularity. You'll learn the advantages of breaking your programs — or projects for that matter — into small manageable pieces.

CHAPTER 3

the memory connection



Random Access Memory can hold information that looks like a program to a microprocessor such as the 6809. Or, it can hold information that is manipulated by a program. These two types of information rarely get confused on a computer running OS-9 because they reside in different parts of memory.

If you run the OS-9 MDIR utility command and take a close look at the program memory area in your OS-9 based computer, you'll see a bunch of names. These names identify a number of small programs. All of these programs live in modules. In this chapter we talk about memory modules and other memory related matters, including:

Modularity
OS-9 Memory Use
OS-9 Module Directories
OS-9 Modules
OS-9 Module Headers
Memory Management

MODULARITY

Here's a real world analogy that may help explain the concept of modularity. Close your eyes for a moment and imagine how you would react if someone told you that you had to build a condominium.

Don't panic. Rather, look for a logical solution to the problem. Visit a local construction site and study the activity. You'll probably see large cranes lifting giant pieces, the size of a living room, into place.

The rooms, or modules, have been built somewhere else and carried to the construction site on large trucks. One module might come from a company that specializes in living rooms, another from a firm that builds staircases.

Once the individual pieces arrive on site, they are put together in the proper order. The contractor combines the modules he needs to build the apartment that the customer ordered. The result is an apartment with the best living room and the best staircase that can be built — at least, the most cost effective.

The authors of OS-9 used a similar approach when they wrote this powerful operating system. They used a technique called structured programming.

First, they defined each problem they were trying to solve in terms of a number of smaller problems. Then, they broke the smaller problems down into still smaller problems. Eventually, they reached the point where they could translate each of the smaller problems directly into a piece of code their computer could understand.

Then, they realized that many of the smaller problems they had been solving were similar. They started to save their solutions and use them over and over again. They saved their solutions in program modules. Before long, they had saved enough modules to construct OS-9.

OS-9 is a powerful operating system. Yet, it is not one giant program. Rather, it is a number of small program modules working together in a synergistic manner. That's modularity.

HOW MEMORY IS USED

If you peek into the memory of an OS-9 computer while it is running you will see the names of all the modules that have been loaded at the top of memory. You'll see data at the bottom. The memory in between is free memory that you can use to run additional programs.

There are a couple of requirements that your computer must meet if you plan to run OS-9. First, the address of the first byte of random access memory must be zero — $0000. Second, your memory must be continuous — it must start at $0000 and run through to the top of your memory.

There can be no gaps in your memory. In other words, you could not install a 16K memory board addressed from $0000 to $3FFF and another addressed from $5000 to $8FFF. You would need to start the second block at $4000 so that every byte in your memory map would be accounted for — from $0000 to your top of memory — $7FFF.

If you have a Color Computer, the design engineers at Tandy met this requirement for you. The memory on your Color Computer starts at $0000 and runs continuously to $FEFF when you are running OS-9.

Each time you bring OS-9 to life, it runs a special routine that looks for the end of your memory. This operation is automatic and you need not worry about it. However, OS-9 needs the information gathered by this routine so that it can reserve a few blocks of memory for its own use.

Actually, OS-9 sets aside two areas of memory. First, it reserves about 1,000 bytes at the bottom of memory. The Kernal uses this area as a workspace to hold temporary information.

OS-9 sets aside another block of memory at the top of your computers memory map. It needs this space for the buffers it uses to read and write information to the disk drives, printers and other devices connected to your computer.

TOP DOWN LOADING

The above heading has nothing to do with the "top down programming" techniques you may have studied in a book or magazine. Rather, it tells you how OS-9 loads your programs when you need to run them.

If you type a command on your keyboard that names a program that is not resident in memory, OS-9 tries to load it from your current execution directory. If it finds the program in this directory, OS-9 writes your new program module into the very top of the free memory space in your computer.

If you are running your first program, OS-9 will load it just below the buffer area that it reserved for itself when you first started it. Later if you load or run another program, it will be written into memory directly below the first program you loaded.

We mentioned earlier that the variable information used by an OS-9 program is kept separate from the program itself. Since this data memory area begins at approximately $0400 hexadecimal, your very first program will most likely have its data stored beginning at $0400.

Later, if you start another program running, that program's data memory area should start just above the end of the memory area used by your first program. You should be aware of one additional point, however. All data memory areas — and all program modules for that matter — start on an even page boundary.

Let's use an example to explain. Suppose that the first pro-

gram you load uses just under one page of memory — 249 decimal or $EF bytes hexadecimal. Further, let's assume that the data memory area used by your program starts at $0400. That means its data area would end at $04EF.

Where do you think the data memory area would start for the next program you run, $04F0? Wrong! It would start at the next even page boundary or $0500. This happens because OS-9 gives your program memory one page, or 256 bytes, at a time.

EVERY MODULE HAS A NAME

Just as every person has a name, every OS-9 program has a name. When your boss needs to tell you what to do, he addresses you by name. If you expect OS-9 to run a program for you, it must be able to call that program by name.

The telephone company gives you a directory so that when you need to call someone, you can find their number. OS-9 has a directory too — a directory that contains the name of each program resident in your computer's memory. OS-9 may have loaded these programs automatically when you brought it to life, or you may have loaded them. OS-9 calls this directory a module directory.

Each time you run the OS-9 MDIR utility command, you will see the names of the programs stored in your memory. You can learn a lot more about the programs in the module directory if you type an ' e ' — for extended — on your MDIR command line. For example, this extended report can tell you the actual memory address where each module is stored and how many processes are using a module.

This use count also tells OS-9 when a module is not being used. This is important because when a module is not being used it can be discarded and its memory released for other programs.

Here's how it works. When a program needs to use the code or data in a module, it links to it. When it does this, OS-9 automatically increases the module's link count by one. When the program is finished with the module it unlinks from it and OS-9 decreases the link count by one. When this link count reaches zero, OS-9 knows that the module is not being used and removes it from the module directory, releasing the memory so that it can be used by other programs.

OS-9 manages your memory for you automatically — you don't need to do a thing. Since this memory management is totally transparent to you, you will never see anything happen either.

OS-9 calls the memory located between the top of the data memory area used by the last program you ran and the bottom of the last program module itself, free memory. It uses this free

memory when it is needed. When does OS-9 need memory? Let's try to make a complicated answer simple.

First, OS-9 will need to use some of your free memory when you load a new program module. Then later when you run this program, OS-9 will need to give the process you have started some memory to use for its data memory area.

Sometimes a program itself may request more memory. OS-9 will give a program more memory if it is available. And finally, OS-9 often needs additional memory to use for a buffer when it opens an input/output path to a new device or file for you.

The process is reversed and OS-9 puts memory back into the free memory area when a program stops running or when you unlink — or unload — a program module.

This magical give and take is made possible by information contained within each OS-9 module. You cannot load anything into the memory of an OS-9 computer unless it is in a module. Further, all modules follow a precise format so that OS-9 will know where to find specific information within a module.

For example, if OS-9 needs to know the size of a module, it will always look at the third and fourth bytes of the module's header.

WHAT DOES A MODULE LOOK LIKE

Let's play a guessing game. If you were an OS-9 module, what do you think you would look like? First, you would have three parts — a header, a body and a CRC.

The header is like the preface in a book or the lead sentence in a newspaper story. By looking at it you — or OS-9 — can find out everything you ever wanted to know about a module. For example, you can find out how many bytes are stored in a module, how much memory a module needs for data storage, and a module's exact location in memory by looking at the module header.

Additionally, the module header can even tell you what type of data is stored in a module. Specifically, the type byte tells OS-9 if the module contains a program, a subroutine, a device descriptor, a device driver, a file manager or just plain data. All of this information is stored in the most significant four bits of the type byte in the module header.

The least significant four bits of this same byte tells OS-9 which language uses the module. A module could contain BASIC09 intermediate-code, COBOL intermediate-code, FORTRAN intermediate-code, 6809 object code or PASCAL pseudo-code. Each higher level language that runs under OS-9 checks the type byte before it tries to run the code in any module. Why? Just imagine what would happen if BASIC09 tried to run PASCAL p-code. The results would

most likely be less than optimal.

Another byte in the module header tells OS-9 if more than one person can run the code stored in a module at the same time. A module of this type is called sharable. Technically, a sharable module contains "reentrant" code.

Come to think of it, why would OS-9 want to let more than one person run the code in a module at the same time? Think about it for a minute and you'll see how sharable modules can save a lot of memory.

On older operating systems two copies of a BASIC interpreter needed to be in memory if two users wanted to run a BASIC program. Effectively, this meant that two users could not run a program at the same time on small 64K microcomputers. Not so with OS-9. BASIC09, which is nearly 22 thousand bytes long, can be used by several users at the same time because it is reentrant and each user is assigned to a separate data memory area.

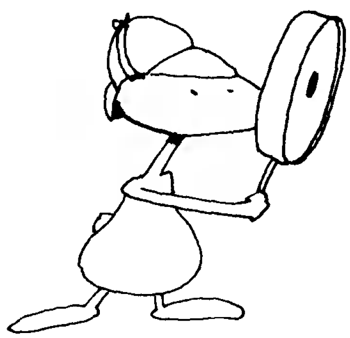
REVISION NUMBER IS VERY IMPORTANT

The least significant four bits of the "sharable" byte in a module header holds the revision number of the module. The lowest possible module revision number is zero. The highest is 15.

The revision number of a module is very important because it determines which module stays in memory when you try to load two modules of the same type that happen to have the same name. Here's the rule.

OS-9 only keeps the module with the highest revision number in its module directory. You'll find this handy when you need to replace code stored in a ROM. To run a later version of the code in ROM, you need only load a module with the same name, but a higher revision number, into RAM memory.

HOW OS-9 FINDS A MODULE'S NAME



OS-9 finds the name of a module by looking at the fifth and sixth bytes in the module header. These bytes contain a value which tells OS-9 the distance between the beginning of the module and its name string.

The first nine bytes of all module headers always contain the same information in exactly the same place. Most module headers store more information in the bytes that follow. The number of additional bytes and their specific location is determined by the module type.

The chart below shows a typical module header. Headers generated by ASM, the OS-9 assembler and BASIC09 both look like this. Our table shows you the precise location of the information.

DISTANCE FROM START OF MODULE	PURPOSE OF INFORMATION	OFFICIAL NAME
0	marks the beginning of the module	Sync Bytes
2	gives length of module	Module Size
4	points to location of module's name	Name Offset
6	reports type of module and language that uses it	Type/Language
7	contains attributes and revision number of module	Attributes/ Revision
8	Contains the cyclic redundancy check (CRC) of the module header	Header Parity Check
9	gives distance between module beginning and actual program code	Execution Offset
11	tells how much data memory area the program needs	Storage Size

You can use Ident, an OS-9 utility command, to read the information stored in a module header. Here's a look at an "ident" of the module, "ds" — the screen editor we are using as we write this book.

```
Header for:     DS
Module size:    $38F3      #14579
Module CRC:     $A3C080    (Good)
Hdr parity:     $FD
Exec. off:      $000D      #13
Data Size:      $7FFF      #32767
Edition:        $1E        #30
Ty/La At/Rv:    $11        $81
Prog mod, 6809 obj, re-en, R/O
```

HOW DOES OS-9 FIND ITS MODULES

OS-9 uses the sync bytes in the first two bytes of the module header to find all valid modules when it starts up. The two bytes, $87 and $CD, hexadecimal are both unused 6809 opcodes.

When OS-9 finds these two bytes together it suspects that it

has found the start of a bona-fide module. To find out for sure, it checks the CRC of the module header. If the module header passes the test, OS-9 reads the module size from the header and runs a Cyclic Redundancy Check on the entire module. If it passes, the module is placed in the module directory.

The body of a module follows the header. It usually holds pure code that is actually run by OS-9. Sometimes however, it contains only data that can be read by other programs.

Following the last byte of the program section of an OS-9 module you will always find three additional bytes. They contain the result of a Cyclic Redundancy Check. Hence, they are named CRC.

OS-9 uses this CRC value when it loads a module into memory. The value read from the last three bytes of the disk file must equal the value computed from the code that was loaded into memory. If they do not agree, then there has been a load error and OS-9 will not use the code in the module.

ADDITIONAL RULES

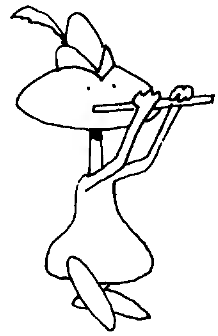
Two additional rules govern OS-9 memory modules. First, the code contained in a module must be position independent. This is required because OS-9 always writes a module into the next available free memory area when it is loaded from a disk file. You never know where a program is going to be loaded ahead of time. And second, the program in the module cannot modify itself. All changing data must be stored in another area of memory.

STANDARD MODULES FOUND IN COLOR COMPUTER OS-9

Major modules in your Color Computer's OS-9 operating system include:

OS9	
OS9p2:	The kernal which forms the heart of the operating system.
CLOCK:	Divides the 6809's time between several different jobs or processes by managing interrupts from the 60 cycle power line. This makes the processes appear to be running at the same time. It also keeps the time of day.
IOMan:	Manages all requests for Input/Output from all devices.
RBF:	Manages all random files stored on floppy disks and other block oriented devices. It also handles all directories and other file information.

SCF: Manages all communication with your keyboard, your Color Computer screen and other devices that read or write one character at a time.



PIPEMAN: Manages all communication between processes using an OS-9 pipeline. It uses a device driver named PIPER.

CCDISK: A device driver that sends and receives data to your disk drives. It works with the RBF file manager and the /D0, /D1 and /D2 device descriptors.

CCIO: A device driver that communicates with the hardware inside CoCo. It services both the keyboard, and the screen. It works with the SCF file manager.

RS232: A device driver for the RS-232 port. You use it when you timeshare your CoCo with an external terminal. It also works with the SCF file manager.

PRINTER: A device driver for the printer port on the rear of your CoCo. You can only output data with this driver.

TERM: A device descriptor that contains data used to tell your programs what your terminal looks like. It works with the CCIO device driver.

T1: Another device descriptor that tells your programs what they can expect from your external terminal. It uses the RS232 device driver.

P: A device descriptor that describes your printer. It works with the device driver named PRINTER.

D0: A device descriptor that contains the size and format of the disk drive number zero. It is used by the driver, CCDISK. You can read and write to additional drives with device descriptors named /D1, /D2, and /D3.

Most standard OS-9 systems contain the following modules in addition to those listed above.

ACIA: A device driver that communicates with the standard Asynchronous Communications Interface Adapters used in the serial ports of most computers.

PIA: A device driver that communicates with the Parallel Interface Adapters used in most parallel ports.

You can tell which modules are in memory immediately after you start your OS-9 system by running the MDIR utility.

SUMMARY

In this chapter you have learned how OS-9 uses modules to manage the memory in your computer. We've also explored the information you can find in an OS-9 module header.

In Chapter 4 we'll look at the OS-9 Software Connection and introduce you to both the Kernal and the Shell.

CHAPTER 4

the software connection

An OS-9 computer has four major software components — a Kernal, a Shell, a set of utility programs and a set of application programs. In this chapter we'll show you how they all work together as we introduce you to:

The OS-9 Kernal
OS-9 Processes
OS-9 System Calls
The OS-9 Shell
The OS-9 Utility Command Set
OS-9 Application Programs

AN OVERVIEW

The Kernal is the heart of your OS-9 computer. Yet, it's a lot like a servant. The other parts of OS-9 call on the Kernal to perform a variety of services. On the Color Computer the OS-9 Kernal comes in two parts, OS9 and OS9p2.

The Shell is the command interpreter for the OS-9 system. It accepts your commands and makes sure that they are carried out. In this manner, it is a lot like a sergeant in the Army or a chief petty officer in the Navy.

When you type an OS-9 command line, you play the part of an officer giving an order to the Chief. The Chief — the OS-9 Shell —translates your command into a language that the troops — the OS-9 Kernal — can understand. As a result, the job gets done.

Nearly 50 utility programs — or commands — are supplied with the Color Computer version of OS-9. These commands do

many things. For example, they copy files, edit text and list directories. We take you on a tour of the complete OS-9 Utility Command Set starting in Chapter 10.

You can develop your own application programs using tools such as ASM, the 6809 assembler in the OS-9 Utility Command Set, and high level languages like BASIC09 and C that run on OS-9.

THE KERNAL

While the Kernal works like a servant in some ways, it acts like the boss in others. Although it performs every task requested by the other parts of OS-9, it also supervises the operation of your computer and manages its resources.

On the Color Computer, you will find the Kernal stored in the top 4,000 bytes of memory. It starts at $F000 and ends just below the input/output hardware at $FF00. On most other computers the Kernal is stored on two 2K ROMS which load somewhere near the top of memory.

The Kernal has many jobs. First, it gets OS-9 ready to run when you boot your system. In programmer parlance, it initializes your system. Then, once OS-9 is on the air, it does the many jobs the other OS-9 parts ask it to do.

Additionally, the Kernal manages your memory, manages your CPU time and processes interrupts. About the only thing it doesn't do is windows — and input and output. When the Kernal receives a request for input or output services, it passes the job on to another OS-9 module named IOMAN. IOMAN stands for input/output manager.

INITIALIZATION

Each time you start up or "boot" your OS-9 computer, the Kernal gets it ready to go. First, it searches through every byte of memory in your computer to see if it can find any valid OS-9 modules. On many OS-9 computers, it will find several modules stored in ROM. Since the information stored in ROM memory does not go away when the power is turned off, these modules are always there, ready to use when the computer is turned on. If the Kernal finds any modules, it puts them in the module directory. Then, it runs a quick check to see how much RAM memory is available and reserves some space for its own use.

Next, the Kernal looks for the rest of the modules that OS-9 needs and loads them from a disk file named OS9Boot. After this file is loaded, the Kernal puts the names of all the modules it found in OS9Boot in the module directory. Once they are loaded, the Kernal treats these modules as ROM. This means that you cannot remove them from memory by unlinking them. After the module

directory has been brought up to date, the Kernal completes the initialization process by running a program named SYSGO. SYSGO starts OS-9 running on your computer.

A PROCESS IS A PROGRAM

OS-9 can do more than one thing at a time. This is possible because the Kernal uses a technique called multiprogramming or multitasking.

The basic unit of multitasking is the process. A process is simply a program doing work. When you run an OS-9 program, the program you are running becomes a process.

Generally speaking, a process does things. For example, while you are running the OS-9 list command, you could say "this process is listing the contents of my file on the terminal." You cannot get any work done on an OS-9 system without creating a process to do the job.

Since your OS-9 computer only has one CPU, a 6809 microprocessor, it actually can only work on one process at a time. However, it creates the illusion that is doing a lot of things at the same time by switching the 6809 from one process to another many times each second. Since the 6809 can execute several million instructions each second, it appears to make quite a bit of progress on each job — even in a second.

OS-9 uses a technique known as timeslicing to share the 6809 microprocessor's time with each program — or process — running on your computer. It works like this.

A hardware real time clock interrupts the 6809 60 times each second. Each one of these interrupts is called a tick. The time between ticks is known as a time slice. On your Color Computer the interrupts come at the power line frequency or 60 times each second. Since there are are 60 ticks each second, a time slice on your Color Computer is 1/60 th of a second.

PROCESSES CAN BE IN ONE OF THREE STATES

You will always find a process in one of three states. It may be active — it is actually doing something. Or, it may be waiting or sleeping.

OS-9 lets an active process have its share of your 6809's time. It does not give any time to processes that are waiting or sleeping.

Processes in a waiting status are on hold. They just sit in memory until another process is finished. For example, when you run a utility command program sequentially, the Shell runs it immediately and then waits for it to finish. While your utility command program is running, the Shell is in the wait state. While it is in

this wait state, it doesn't do anything.

A process that is sleeping has checked out. It has placed itself on hold for a specified length of time. A process comes out of this self-imposed sleep and goes back to work — becomes active — when the proper length of time has passed or when it receives a signal telling it to go back to work.

Processes are a lot like people. They need to talk to each other. They talk to each other using signals.

ALL ABOUT SIGNALS

Just as the signals you see along a railroad track keep the trains from running into each other, the signals in an OS-9 computer keep processes from running into each other.

Here's another use for a signal. A process that is sending data to a printer often puts itself to sleep while it is waiting for a signal that tells it the printer is ready for another character. The signal in this case comes from the printer hardware. It is relayed to the Kernal by the device driver module.

Signals give OS-9 a way to talk to each other. Stated another way, they coordinate communication between processes. Signals work a lot like software interrupts — they suspend a process, execute a routine and then return to the suspended process. A process sends a signal to another process by sending a service request to the Kernal.

When a process sends a signal to another process, the signal carries status information in a one byte code. Up to 256 distinct codes can be sent to a process. All of them except four are defined by the programmer.

The four predefined signal codes kill a process, wakeup a sleeping process, signal that an operator has aborted the process from the keyboard or, signal that the keyboard operator has interrupted the process. The table below shows the value of these codes and the corresponding result.

VALUE	RESULT
0	Kill or Abort the process receiving the signal
1	Wakeup a sleeping process
2	Abort the process upon request of the Keyboard Operator
3	Interrupt the process upon request of the keyboard operator

You can define all other values between 4 and 255 and use them to send special signals to your own processes. And, you can set a special signal trap that insures that an operator cannot kill one of your processes.

A signal trap sends your program to a routine that contains

instructions that tell it what to do when it receives a signal. You set a signal trap within your assembly language programs by executing an intercept service request.

It is very important that you add a signal trap to your programs because if an operator sends a signal — a keyboard abort for example — to a program that does not contain a signal trap, OS-9 will abort your program whether you want it to or not.

SERVICE REQUESTS TELL THE KERNAL WHAT YOU WANT

When you go to a service station, you ask the attendant to fill your tank. Essentially, you have requested that he perform a service for you.

Likewise, you must issue a request for service to the Kernal when you want OS-9 to do a job for you. OS-9 programmers call these service requests "system calls."

There are two types of system call. The first deals with requests for input or output. For example, when you request a character from your keyboard, you are using a system call named I$Read. Other input/output service requests are used to attach devices, open and close paths, delete files, make directories, etc.

The second type of system call is known as a User Mode Service Request. You use this type of system call when you want the Kernal to do something for you that has nothing to do with input or output. For example, when you need to load a module from a file on one of your disk drives, you issue the F$Load service request.

You can tell that a request is an input/output service request by looking at its name. The mnemonic of all input/output service requests begins with I$. For example, you use I$Close to close a path to a device or file.

Likewise, the names of all user mode system requests begins with an F$. For example, if you need to link to a memory module, you issue a service request named F$Link. All OS-9 service requests are defined in the OS9Defs file in the DEFS directory. The OS9Defs file makes it easy to write OS-9 assembly code.

For example, if you want to read a byte from the keyboard or a disk file, you need only load a few of the registers in the 6809 with the proper codes and then issue an I$Read service request. Specifically, you need to load the A-register with the path number, load the X-register with the address where you want to store your data and load the Y-register with the number of bytes you want to read.

After you have loaded the registers, you make the actual service request by using the special OS-9 assembler operator,

OS9. Here's what the code would like in an assembler source code file.

```
LDA PathNum
LEAX Buffer,U
LDY #1
OS9 I$Read
BCS Error
...
```

If OS-9 was able to read the byte you requested it will return from the I$Read service request with the carry bit in the 6809 clear. If there has been a problem, it will set the carry bit and place an appropriate error code in the B-register. For this reason, you will generally see a 6809 *branch if carry set* (BCS) mnemonic following each service request in an OS-9 program. The BCS instruction sends your program to a special error handling routine.

All OS-9 service requests are documented in your OS-9 System Programmer's Manual.

PROCESSES ARE EASY TO CREATE

You'll find that it is very easy to start a new process when you write an OS-9 program. In fact, the Kernal does most of the job for you automatically.

To create a new process, you request an F$Fork system call. When you make this request you must also tell the Kernal the name of the program you want to start by giving it the name of the module that contains the program as a parameter. You may also give it file names or other information for the new process to use when you issue the F$Fork service request.

The Kernal first tries to find a module with the name you gave it in the module directory. Remember, this directory contains the name of all modules that are present in memory. If it finds the name of your program in the module directory, the Kernal will link to the module and run it for you.

If the Kernal cannot find the name of your program in the module directory, it looks for a file by the same name in your current execution directory — usually, /d0/CMDS. If it finds the file, it will load it into memory, link to it and run your program.

When the Kernal links to your module, it creates a process descriptor that holds a lot of information about your program. OS-9 uses this process descriptor to keep track of the state of your process, its priority, the memory it is using, etc.

The Kernal then sets aside an area of memory that your program can use for data storage. It finds out how much memory your

program needs by reading the storage size value from the module header.

When the Kernal starts a new process, it assigns it a unique ID number. These ID numbers can range from one to 255.

If the Kernal cannot complete any one of the steps above, it aborts and does not create your process. Yet, it won't leave you hanging. It let's you know what happened by printing a message that contains a special error number that tells you what went wrong.

PROCESSES — LIKE PEOPLE — HAVE MANY CHARACTERISTICS

Processes are a lot like people. In fact if you think of OS-9 as a family of processes, you will find it much easier to understand. Let's look at the genealogy of a family of OS-9 processes.

When a process creates another process, it becomes a parent. The new process is called a child. Further, if the child creates another process, it also becomes a parent. A process can create any number of children.

This whole discussion may seem absurd. Yet, the family concept makes OS-9 much easier to understand. If you apply it when you look at the output of the OS-9 PROCS utility, you can almost visualize a family tree.

CHILD PROCESSES INHERIT PROPERTIES FROM THEIR PARENTS

Just as human children inherit characteristics — brown hair, blue eyes, etc. — from their parents, OS-9 child processes inherit a number of properties from their parent process.

For example, each person using a computer running OS-9 has been assigned a User Number. If a person starts a process, that process belongs to him — it carries his user number. If that process then starts another process, the child process inherits his user number and belongs to him also.

Other properties that are inherited by a child process include the standard input and output paths, the process priority, and the current execution and data directories.

For example, if the standard input and output path used by a parent process is sending data to the main terminal device, /TERM, any children created by the process will also send their data to /TERM.

Likewise, if you start a process with a low priority, any children created by that process will also have a low priority. This is important because the process priority tells the 6809 how important a job is to you. If you give a process a low priority, the 6809 will give it

very little time.

YOU TALK TO OS-9 THROUGH THE SHELL

As you begin to use OS-9, you will find yourself carrying out a dialogue with the Shell. After a little practice, it should become meaningful.

When you type a command on your keyboard, you are talking to the Shell. The Shell is a command interpreter that translates the words you type into a language the Kernal can understand. The Kernal translates the command from the Shell into an action inside your computer.

You will know when you are talking to the Shell because you will see this prompt on your screen.

OS9:

When you see this prompt, you'll know that the Shell is waiting for you to enter a command. To answer the prompt, you simply type a command line followed by a carriage return. You can use lowercase letters, uppercase letters or a combination of the two when you type a command — the Shell doesn't care.

The Shell itself does not carry out the commands you give it. Rather, it looks at your command line and starts a process using the name of the OS-9 application or utility you typed in your command line. The utility command does the actual work.

It is easy for the Shell to find the right utility command because the name of the command you type is usually the same as the name of the module in memory that contains the command.

But, when the module is not already in memory, the name of the command is almost always the same as the name of the file that holds the module in your current execution directory. More than 50 utility commands come with the OS-9 operating system.

You'll find that the form of the commands you type rarely changes. Only the command names will change. Your command line will usually contain the name of a command, various options that give it special information and an argument that tells it where to find its data. That argument is most often a filename.

EXAMINING THE OS-9 COMMAND LINE

When you take a closer look at an OS-9 command line you will notice that the first thing following the prompt is the name of a program — either one of the OS-9 utility commands or an application program of your own. It can be the name of a program located in a module in memory or the name of a file that holds the program on a floppy disk.

The module can contain 6809 machine code that executes directly, compiled intermediate code from a higher level language like BASIC09 or PASCAL or a procedure file. The type of code it contains determines exactly what will happen when you type <ENTER>. Here's what happens when you type a command line.

If the Shell finds a module in memory with the name you have typed, it will run the program. If it doesn't find the program in memory, it looks for a disk file with that name in your current execution directory. If it finds the file, it loads it into memory and runs it.

If the name you typed is not the name of a module in memory or the name of a file stored in your current execution directory, you still have one more chance — it may be the name of a procedure file. The Shell knows this, and searches your current data directory for a file with the same name.

If the Shell finds a file in your current data directory, it treats it as a procedure file. A procedure file is similar to a Shellscript used on a UNIX-based computer.

Instead of holding 6809 object code that runs in your computer, or I-code that is run by a high level language interpreter, a procedure file contains a text file that looks much like a series of command lines that you would type from the keyboard.

When the SHELL runs a procedure file, it reads the text one line at a time — just as if it were reading a command line from your keyboard. Then, it runs that command. After it runs the first command line in the text file, it runs the second, etc. It repeats the process until it reaches the end of your procedure file.

PASS THE WORD — USE A PARAMETER

The command name the Shell reads from either your keyboard or a procedure file is usually followed by additional names. We call these additional names parameters. Let's take time now to define this word.

If you want to succeed in business, you must pass the word to your employees. Likewise if you want an OS-9 program to run properly, you must pass the proper parameters.

A parameter can be a single character or a string of characters typed behind the command name in an OS-9 command line.

As you can see, a parameter gives directions to a program. It is separated from the program name by one or more spaces. For example, if you want to list a file named "Rainbow" to your terminal, you must type:

OS9:list Rainbow <ENTER>

If you want a "hardcopy" of the same file you can type:

OS9:list Rainbow >/p <ENTER>

Or, you can even send the listing to another file:

OS9:list Rainbow >SonOfRainbow <ENTER>

OS-9 is indeed a very versatile operating system.

Sometimes the parameters in your command line are options, or modifiers. For example when you want to list the names of the files in your current data directory to your Color Computer screen, you type:

OS9:dir <ENTER>

To get more information about your files, You can type:

OS9:dir e <ENTER>

This command gives you all the available facts about each file in your current data directory.

The ' e ' is an option. It means list the "entire" directory record. Speaking of directories and options, if you would rather see the names of the files stored in your current execution directory, type this command line:

OS9:dir x <ENTER>

If you want to see all available information about the files stored in your current execution directory, type:

OS9:dir x e <ENTER>

You can run OS-9 programs several ways. You can run them sequentially — one after the other; you can run them concurrently — all at the same time; or, you can pipe them. When you do this, the output of one feeds the input of another.

There are two ways to run programs sequentially. The most obvious way it to type one command line followed by a carriage return, wait for the program to finish and then type the next command line.

Or, you may type more than one command on a line. To do this, you use a semicolon to separate the commands. Here's an example:

OS9: copy hisfile herfile ; dir >/p <ENTER>

This command copies the file named "hisfile" from the current

data directory to a file named "herfile" in the same directory. When that job is finished, it will immediately send a listing of the files in your current data directory to your printer.

If you want to run more than one program at the same time you may ask OS-9 to run your programs at the same time — concurrently — by typing an ampersand, ' & '.

Within reason, you can run any number of programs at the same time. The amount of memory in your computer is the major limiting factor.

Pretend for a moment that you have just finished an assignment in school. You need to print it so you may turn it in to your instructor, but at the same time you need to be working on another term paper. To do both jobs at the same time, try this!

OS9:list EnglishII.Assignment >/p&
&004
OS9:edit Term.Paper_History

The printer will start immediately after you hit the <ENTER> key. Yet, the familiar OS-9 prompt will appear on your screen almost instantly. When it does, you can type the second command line and start writing. The printer will run as long as it takes to print the English assignment. On most computers it won't slow down your editing at all.

YOU CAN TELL OS-9 HOW MUCH MEMORY YOU NEED

Some OS-9 programs need very little memory to run. Others require thousands of bytes. This is not really a problem because each program module header tells OS-9 the minimum amount of memory needed to run the program. However, when you need more memory, it is an easy matter to request more with the OS-9 memory size modifer. There are two ways you can do this.

OS9:copy #8 myfile yourfile

This command line tells the OS-9 copy utility to use eight, 256-byte pages of memory — a total of 2048 bytes. Let's try another.

OS9:copy #2K hisfile herfile

Believe it or not, this command line also gives the copy command 2048 bytes of memory. The difference? It requests two ' K ' or two thousand bytes of memory.

LOGGING ON A TIMESHARING TERMINAL

With OS-9 you can do more than just print one file while you are editing another. In fact, one of the most important things you

can do on a computer that can run more than one program at the same time is let two or more terminals share the computer. On many OS-9 computers, several different users can work on different terminals at the same time.

For example, you can use your editor and Color Computer screen to write a news release about a new product while your spouse runs a BASIC09 program to balance the checkbook on another terminal. Just plug a terminal into the RS-232 jack and type the following:

ON THE COLOR COMPUTER:

OS9:tsmon /t1&
&005
OS9:

YOU'LL SEE THIS ON THE OTHER TERMINAL

OS-9 Level I Version 1.0 Timesharing System 9/1/84 12:30:45

User name?: esther
Password:

Process #5 logged 9/1/84 12:31:05

Shell

OS9:

Your Color Computer prompted you to go back to work immediately. But while one of you is writing that news release, the other can be working on the checkbook.

When you first run the timesharing monitor program, TSMON, nothing happens. The remote terminal remains idle. TSMON simply sits and waits for someone to hit its return key.

Also, when you want to work from an additional terminal you must log on. To do this you simply type your name and the proper password. You will need to give everyone in the family a password before they try to log on, because if they do not type the proper password, OS-9 will not let them compute.

When you have finished your work on a remote terminal, you will want to log off. To do this, simply hit the ESCAPE key while the Shell is waiting for a command. When you hit the <ESCAPE> key, an end-of-file signal is sent to TSMON. The terminal screen will go blank and TSMON will again sit and wait for someone else to come along and hit the <ENTER> key.

A terminal used for timesharing on your Color Computer is limited to 300 Baud by the hardware. A "bit banger" port that uses

software to simulate a serial port, and a keyboard without an encoder that requires the 6809 to constantly scan the keys, burns up a tremendous amount of the 6809's time. This processing load, combined with an interrupt rate based on the power line frequency, does not leave enough time for you to timeshare at a faster Baud rate.

SUMMARY

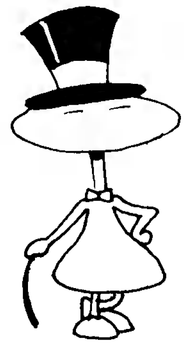
In this chapter you have been introduced to three major OS-9 software components — the Kernal, the Shell and the utility command set. You've learned that processes are programs doing work. And, you have been introduced to signals and OS-9 System Calls.

Take a quick breather now, then join us in Chapter 5 as we take a close look at the OS-9 File System.

CHAPTER 5

the file connection

OS-9 FILES HOLD YOUR DATA



The OS-9 file system is so advanced that you will soon find it is very easy to manage complex information. In this chapter we talk about the hierarchical file structure that makes it happen and tell you about the security system that protects your data.

We'll talk about the "unified input/output system" that Microware developed for OS-9. Their approach makes any piece of hardware you connect to your computer look the same to your programs. And, we'll talk about your current data and execution directories.

DIRECTORIES HELP YOU ORGANIZE YOUR FILES

Files make OS-9 data storage possible. They hold business data such as payrolls and spreadsheets. They hold word processing documents like letters and reports. In fact, they even hold the programs you use to manipulate the information stored in your data and word processing files.

OS-9 files are a lot like file folders in a file cabinet — they will hold anything you put in them. And just as every file folder has a label, every OS-9 file has a name so that you can find it later.

An OS-9 directory is a special file that contains a list of file names rather than data. It works a lot like a telephone directory. When you tell OS-9 you need to use a file, it looks in a directory to find its location on your disk drive.

OS-9 directories give you a way to organize your files. Think of them as the drawers in a file cabinet. Just like their paper counterparts, OS-9 directories give you a way to gather related files in a common place so you can find them easily.

If you take a close look at the OS-9 file structure, you will see how it can help you organize your office.

How is your office set up? In most organizations, each person has a desk and an individual file cabinet. That cabinet is divided into drawers which contain information relevant to a specific part of that person's job.

How can we automate that office? Why not start with a microcomputer that uses an operating system that lets several people work on different terminals at the same time? OS-9 can do the job.

If we were to use one of the older operating systems, CP/M or FLEX for example, everyone's files would be in the same directory — in the same file cabinet so to speak. When a secretary needed to find a file for her boss, she would have to look through a list of every file on the hard disk. It would be like looking at every folder in every drawer of a large file cabinet. It would take a long time. The boss would get angry.

So much for the older operating systems. Now lets see how OS-9, with its multiple directories, can help solve the problem. We'll go back to the same office, with the same hard disk. But, this time we'll pretend that OS-9 has been installed.

When we look at the computer on our second visit we will notice that the hard drive now has a name — probably something like /H0. When the secretary asks for a listing of files on the drive —a DIR — it will look something like this.

```
OS9: dir /h0 <ENTER>
DIRECTORY OF /H0 12:22:41

OS9Boot  CMDS       SYS    DEFS
Startup  Read.This  BOSS   SECRETARY
SAM      JOE        SALLY  JANE
```

If we ask her to see more details about those files, she can add the ' e ' option to the command line. When she does this, we will notice that each of the filenames that is printed in all capital letters has a ' d ' printed in a column under the word Attributes. This means it is a directory. You do not need to use all capital letters when you name a directory, but it certainly can make life easier down the road.

Remember, when you create a new OS-9 directory with the MAKDIR utility command, always type the directory name with all capital letters. Then later, when you scan a directory listing you

can immediately separate the directories from the regular files. It should save a lot of head scratching.

Let's take a closer look at the OS-9 directory system. Imagine that the boss needs to see a letter that Sally mailed to his ad agency. If the office computer is set up properly it should be easy for the office manager to find the letter. In fact, she would need only to sit down at the terminal and type:

OS9: chd . ./SALLY
OS9: dir

At this point, she might see something like this:

DIRECTORY OF . 12:22:45

```
ADVERTISING   NEWSLETTER          PAYROLL
SCHEDULES     ACCOUNTS.PAYABLE
```

Since she knows that each of the filenames listed are directories, because they are printed with all capital letters, she would then type:

OS9: dir advertising

She will see something like this:

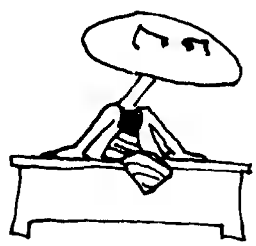
DIRECTORY OF advertising 12:23:41

```
COPY            IDEAS          Agency.Letter
Instructions
```

I'll bet the file "Agency.Letter" contains the letter the boss needs. The office manager can now type the following command to print a new copy for the boss.

OS9: list advertising/Agency.Letter >/p

The example above shows how easy it is to find a file when it has been stored in a logical place. The office manager knew that Sally wrote the letter, so she looked in Sally's directory. Further, since she knew the letter was sent to an advertising agency, she looked in Sally's advertising directory. She had the bosses answer in seconds.



If our office manager had been working on a computer running one of the older operating systems she would have needed to look through hundreds of filenames on a single hard disk directory. It would be like searching through an overstuffed file drawer and would have taken a lot of time. She would have been in a lot of trouble with the boss. The logical, structured approach of OS-9 files saved her day.

The OS-9 file structure is hierarchical. It lets you group similar files in a single directory. And, it lets you group like directories in other directories. This lets you organize your information.

If you turn your imagination loose, you can picture the OS-9 file system as an upside-down tree. In your picture you would see the roots of the tree at the top. Likewise, on an OS-9 disk you will always find the "root" directory at the top of the file structure.

In fact, each disk drive in your OS-9 based computer has a master directory which is called the "root" directory. That root directory is at the top of the file structure on each drive. It contains the names of each file or sub-directory stored on the disk mounted in the drive. A root directory is created automatically when a disk is initialized with the OS-9 FORMAT command.

PATHLISTS HELP OS-9 FIND YOUR FILES

For every OS-9 file there is a pathlist. When the operating system needs to find a file, it follows that pathlist from the root directory to the file. For example, here is the complete pathlist to the letter the office manger printed for her boss.

/H0/SALLY/ADVERTISING/Agency.Letter

We seem to have a paradox. The file structure has made it very easy to organize and retrieve data. But, it could be hard on the fingers. Typing a pathlist like that could get old real quick. Especially if you hunt and peck.

But wait! The office manager didn't type all of that. Rather, she typed:

list advertising/Agency.Letter >/p

Why didn't she need to type /H0/SALLY? That's a good question. And, it gives us a chance to introduce you to the concept of current data and execution directories. Do you remember when the office manager typed:

chd . ./SALLY

When she did this, she told OS-9 to make the directory named SALLY, which was located in the root directory of a device named /H0, the current data directory.

Then, when she typed the LIST command, she used an abbreviated pathlist. Since she didn't need to type a device or root directory name, the list command knew that the file was located in her current data directory — /H0/SALLY.

ABOUT THAT SLASH

How did the system know that she didn't type the name of a device or root directory? Here's the plot.

When you type a slash, ' / ' as the first character of a pathlist, OS-9 looks for the name following the slash in its device directory. This means that if you start a pathlist with a slash, the name following that slash must be a device.

When the first character of a pathlist is NOT a slash, ' / ', OS-9 looks for a file in the current data directory.

In our example above, OS-9 looked in the current data directory — /H0/SALLY — and found the file ADVERTISING. When it looked closer, it learned that ADVERTISING was another directory. It then searched the directory ADVERTISING and found the file "Agency.Letter".

Remember, if you start a pathlist with a slash, you must complete that pathlist. In other words, you must type each name required to trace the pathlist from the device directory through each sub-directory until you arrive at the file you need.

You can save all of this typing by using your current working directories. When you do this, you also speed up your computer. OS-9 will find your files faster because it won't need to search every directory along the road — er, path.

WHY ARE THERE TWO WORKING DIRECTORIES

OS-9 differs from UNIX here. UNIX has one current directory. OS-9 has two — a current data directory and a current execution directory.

This separation lets you store files that contain programs in one directory and files that contain data in another. OS-9 can tell where to look by studying your command line. Here are the rules.

OS-9 looks in the current execution directory when it needs to find a file that holds a program that it needs to run. It assumes that a file holds the program you want to run if you typed its name first on a command line or if you used it in a command line after the word "load." OS-9 uses the current data directory at all other times.

When you start your computer, OS-9 sets the current execution directory to /D0/CMDS and the current data directory to /D0.

ANONYMOUS DIRECTORIES SAVE KEYSTROKES

Sometimes you need to refer to the current data directory, or the next higher-level directory that contains it, but you do not

know the name of either. Of course, you could find out by using the PWD or PXD utility commands, but it's easier to use OS-9's anonymous directories. Remember these shorthand symbols.

A single period refers to the current data directory. Two periods refer to the directory that contains the name of the current data directory.

You can use this shorthand in place of a pathlist or as the first name in a pathlist. For example, the following command lines will all work.

OS9: dir .

This command line tells OS-9 to list the names of all files in your current data directory.

OS9: dir . .

This command lists the names of all files in the directory that contains the name of your current data directory.

OS9: del . ./temp

And finally, this command line tells OS-9 to delete a file named "temp" that is stored in the same directory that contains the name of your current data directory.

If you use these shorthand symbols as part of the first name in a command line, OS-9 will look for the file you name in your current execution directory. Likewise, if you use these shorthand symbols in a pathlist following the load command, OS-9 will look for the file you name in your current execution directory rather than your current data directory.

For example, let's assume that you have stored a group of BASIC09 packed modules in a sub-directory named ICODE in your CMDS directory — i.e., /D0/CMDS/ICODE — and it is your current execution directory. Then, when you need to delete a file named temp in your current data directory you could use the DEL utility command stored in your /D0/CMDS directory by typing:

OS9: . ./del temp

Effectively, you could have done the same thing by typing:

OS9: /D0/CMDS/DEL temp

Which would you rather type?

MORE ABOUT PATHLISTS

But, what do you do when you need to read a file that is not

stored on the same disk drive as your current data directory?

No problem, just enter a complete pathlist. A pathlist is nothing more than a complete description of the route your data must take before it reaches its final destination. OS-9 pathlists may contain the name of a mass storage device, a directory, a file or an Input/Output Device.

For example, many pathlists — like the one used by the office manager above — contain a device name and one or more directory names, as well as the name of a data file. You should note that each name in the pathlist is separated by a slash '/'.

Here are some additional rules you should be familiar with if you plan to type a lot of pathlists. Each name must start with either an uppercase or lowercase letter and may be up to 29 characters long. But, a name isn't required to be that long. You could give a file a name that was only one character long if you wanted to. The trade-off is between typing time now and readability later.

You may also use the numerical characters 0 - 9, the period and the underline symbol in your filenames.

If you're wondering how OS-9 can tell the difference between a filename and a device name, here's the secret.

The name of a device always starts with a slash. If that device can hold multiple files — a disk drive for example — another slash followed by a directory name or filename follows the device name.

When a device cannot handle multiple files — a terminal or printer for example — nothing follows the device name.

OS-9 FILE SECURITY PROTECTS YOUR DATA

If you are using OS-9 on a timesharing computer, you won't need to worry about someone else writing in your data files. OS-9 protects you with its file security system.

Each directory and file has several attributes that tell OS-9 who owns the file and who may use it. Depending on whether these attributes are set or not:

w Only the owner may write to the file
r Only the owner may read the file
e Only the owner may run the program stored in the file
pw Anyone may write to the file
pr Anyone may read the file
pe Anyone may run the program stored in the file
s Only one person may run this code at a time
d You know that the file contains a directory

If a file attribute is set, permission is granted for that attribute. If an attribute is clear, permission is not granted.

For example, consider a file with all attributes cleared except owner read. In this case, only the person who created the file can read it. No one else can read the file. Further, absolutely no one —not even the owner — can write to the file.

Can you see how you can use file attributes to protect your information? Study the section that describes the ATTR command in Chapter 11. Try the sample commands there and you'll understand the OS-9's file security system in no time.

The directory attribute tells OS-9 that a file is a directory file. A directory file is special because it cannot be changed by the user. To change a directory or delete it during an operation would create total havoc with the file system. In fact, there would no longer be a system.

The other file security attributes almost explain themselves. They work because OS-9 automatically stores the user number of the owner of any process that writes a file. If you are the owner of a process, you will own any files that it creates.

If you CREATE a file with none of the public attributes set, you will be the only person — except for the super user — who can read, write or execute that file. The concept of the "superuser" is detailed in Chapter 11.

SUMMARY

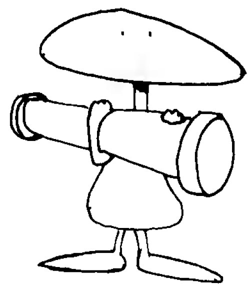
In this chapter we have introduced you to the OS-9 file system. You've learned about directories and files. And, we've shown you how OS-9 file attributes can protect your data secure.

Experiment with the DIRectory list utility now. When you get bored, join us in Chapter 6 where we introduce you to OS-9 input/output.

CHAPTER 6

the outside connection

INPUT/OUTPUT WITH OS-9



If you have grown tired of long chapters, rejoice. This chapter is short. Yet, its fruit is sweet.

We'll introduce you to OS-9's standard input, output and error paths and show you how to redirect these standard paths to other devices or files. By the end of the chapter, you'll need a pipe-wrench as we talk about pipes and show you how to build pipelines.

FOR OS-9 LIFE IS BUT A LONG STREAM OF CHARACTERS

OS-9 will talk to anybody. Or should we say, OS-9 will talk to anything? Since OS-9 treats all hardware devices the same, you can list a file to a modem, a printer, your terminal or another file. OS-9 could care less.

All devices installed in an OS-9 system have a name. And, as we mentioned in Chapter 5, all of these names begin with a slash, '/'.

For example, when you start your OS-9 computer you are usually sitting in front of a terminal. And in almost all cases, when OS-9 comes to life it sets itself up to talk to a device named /TERM. This means that OS-9's standard input path is set up to get characters from the keyboard on your terminal — or /TERM. Its standard output path is set up to send each character to the screen on your terminal — also /TERM. And, a third path called the standard error path, is also set up to send information to the screen on your terminal.

During your session with OS-9 you may need to send a message to someone working on another terminal. One of the most common names for a second terminal on an OS-9 computer is /T1.

In fact, most devices have standard names. For example, when you need to send something to your parallel printer, you usually redirect your output to a device named /P. When you need to send something to your serial printer, you usually redirect your output to a device named /P1. Likewise, most people who install a hard drive on their OS-9 computer name it /H0.

Again, the common thread among these names is the fact that they all begin with a slash, '/'.

When OS-9 sends a character to the screen on your terminal, it does the same thing it would do if it were sending that character to a file on one of your disks. This means that if you write a program that works with a file, it will work with any device. Conversely, if you write a program to work with any device, it will also work with any file.

A program that works like this is file and device independent. Microware calls this approach a "unified input/output system."

Microware's approach to input and output with OS-9 has many benefits. For example, you can copy a file to another file if you need to make a backup copy. Or, you can copy that same file to your screen if you need to take a quick look at it. If you want a more permanent paper copy, you can copy the same file to your printer.

And if the possibilities above don't impress you, consider this: You can also copy a device to a file, or to your terminal, or to your printer, ad infinitum. For example, your terminal is a device that you may want to copy often.

Play around with these command lines. When you want to stop, type <ESCAPE> — <CLEAR><BREAK> on the Color Computer. This will send an EOF signal to COPY and return you to the OS-9 Shell.

OS9: copy /TERM file.from.term <ENTER>

This command line literally copies every character you type on your keyboard to a file named "file.from.term." Incidentally, since you did not type a complete pathlist, that file was stored in your current data directory.

OS9: copy /TERM /TERM <ENTER>

This command line doesn't do much. It copies every character you type on your keyboard to the screen on your terminal. If you have your terminal setup to echo each character from the key-

board to the screen automatically, you may find yourself staring at two copies of each character you type.

OS9: copy /TERM /P <ENTER>

After you type this command line, every character you type will be echoed to your printer. It almost works like a typewriter. But not quite. Did you notice that the OS-9 copy utility does not add linefeed characters when it sends a carriage return. For this reason, you will want to use the OS-9 LIST utility command when you try to copy a text file — or your terminal — to your printer or another terminal. Your command line should look like this.

OS9: list /TERM >/P <ENTER>

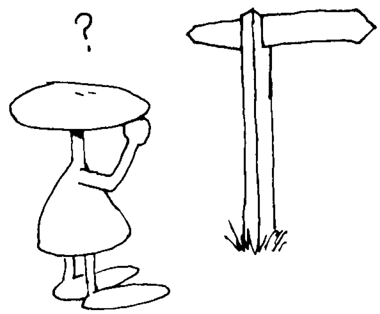
This command line lets you use the keyboard on your terminal and your printer to emulate a typewriter. Here's another experiment for you.

OS9: list file.from.term >/P <ENTER>

Use this command line when you want to print a hardcopy of a file you have stored on a disk. Note, also, that this command line will not work if a file named "file.from.term" is not present in your current data directory.

Every character that is sent — or input — to OS-9 comes to it through a path. Likewise, every character that it sends out to a disk drive, printer or terminal is sent through a path.

Pipelines can carry all types of material — natural gas, oil or water. Likewise, OS-9 input and output paths can carry any type of data. They do not care what it looks like or what it will do when it finally reaches the device at the end of the line.



OS-9 paths can carry the binary code contained in a module saved in a file on a disk to another file on another disk drive. And, they can carry information you have saved in a word processing file to your printer. Again, OS-9 paths don't care what they carry —they just keep the data moving. It is up to the program that uses the information from a path to determine what it means.

STANDARD INPUT AND OUTPUT PATHS

Because of the way OS-9 is designed, most programmers write short programs designed to do one simple task. Further, they make sure that their programs get all of their information from the standard input path. And, when their program sends data to the outside word, they make sure it always uses the standard output path.

The OS-9 Shell itself is a perfect example. It uses OS-9's three standard paths. It reads characters from the standard input path. It

sends characters to the standard output path. And, it sends error messages to the standard error path.

When you type a command on your keyboard, the Shell starts a process using the program that you named in your command line. The program that you are running uses the same standard input and output path as the Shell since it inherits those paths from the Shell.

The Shell normally has its standard input path connected to the keyboard on your terminal. Its standard output and standard error paths normally send information to the screen on your terminal.

When you start to write assembly language or BASIC09 programs for your computer you will learn that these standard input/output paths can also be identified by number. The chart below equates the path number to its function.

PATH NUMBER	FUNCTION
0	Standard Input
1	Standard Output
2	Standard Error Output

STANDARD INPUT AND OUTPUT PATHS CAN BE REDIRECTED

The real power of OS-9 lies in the fact that its standard input and output paths can be redirected. To understand why, we need to look at a typical job performed by a computer.

Imagine that you have been keeping a data file containing information about each check you write during a year. Among other things you include the check number, the name and address of the person receiving the check, the amount of the check, your bank balance after the check has cleared and, finally, the expense category for your income tax records.

Imagine further that you have written a program that lets you list this information on your terminal. You have called it "Who_to" for want of a better name.

"Who_to" works like this. You type the check number; the program prints the name of the person receiving the check and the amount.

If you only need to check one or two checks each month, your program meets your need. But what happens when your business grows and you need a report about 100 checks. Typing in 100 check numbers and waiting for a report on each one could get old real fast.

Enter OS-9 with its ability to redirect the standard input path. Let's see how we can use it.

Why not put a list of check numbers in a file by themselves. That file will contain one check number on each line. Make sure you give the file a useful name — "Check.Numbers" will do nicely.

Once you have created "Check.Numbers" you can put OS-9 and its ability to redirect the standard input path to work with "Who_to". Try a command line like this:

OS9: Who_To <Check.Numbers

This command line will produce a report about each check number in the list you entered in the file, "Check.Numbers". Likewise, you could have "Who_To" print these reports by redirecting OS-9's standard output path to your printer. The command would look like this:

OS9: Who_To <Check.Numbers >/P

Or, you could save your reports in a file.

OS9: Who_To <Check.Numbers >January.Checks

The two magic characters in the command lines above are the less than sign, '<', and the greater than sign, '>'. The first redirects the standard input path. The latter redirects the standard output path.

Remember this!

If you want to redirect the input to a program from another file or device, use the less than symbol, '<'.

If you want to redirect the output of a program to another file or device, use the greater than symbol, '>'.

If you want to redirect the error messages from a program, use two greater than symbols together, ">>".

Or, in table form:

THIS CHARACTER	DOES THIS
<	redirects the standard input path
>	redirects the standard output path
>>	redirects the standard error path

There are many ways to redirect the input of your programs. You may tell your program to get its input from a remote terminal in your workshop. Or, you may tell it to get its input from a modem that uses a telephone line to receive data from a distant keyboard.

Don't forget the output from your program. You may tell your program to send a report to your printer. Or, you may tell it to save the report in a disk file so you can print it later. You are limited only by your imagination.

BUILD PIPELINES TO HELP FILTER YOUR DATA

One of the most important things you can learn from this chapter is to design and write all your programs so that they can be redirected.

In plain English, your programs should read data from the OS-9 standard input path, process it and then write their output to the OS-9 standard output path. Programs that work like this are called filters.

Why are we running this point in the ground? Because, we want you to write your programs this way so that you can use them in a pipeline as you gain experience with OS-9.

"What's a pipeline?" you ask.

Most filters are written to do one simple job extremely well. Their power however, lies in their ability to be combined with other filters in pipelines designed to accomplish a complex task.

Pipelines are built out of pipes. Pipes pass the standard output of one utility command directly to the standard input of another utility command. This makes them especially useful on an OS-9 based computer because you can run more than one process at the same time.

The OS-9 Shell sets up a pipe each time it sees an exclamation point, ' ! ', in a command line. When an exclamation point appears, the standard output from the command on the left side of the ' ! ' is redirected via a pipe to the standard input of the command on the right side of the ' ! '.

To illustrate how pipes work, we will add a few bells and whistles to our program "Who_To".

Earlier we caused Who_To to print a report with information about a number of checks by redirecting the standard input to "Who_To" from a file called Check.Numbers. Now let's show you how you could make it print a sorted report on your printer and save a file for future reference at the same time.

First, we'll use a "sort" utility command to put the list of checks in the file Check.Numbers in order. If we were to run the utility alone, the command line would look like this:

OS9: sort <Check.Numbers

This command line would send a list of check numbers to the standard output path — usually the screen on your terminal. Standing alone, this doesn't do much for you. But, try this.

OS9: list Check.Numbers ! sort ! Who_To

If all goes as planned, you should see a report about each check in the file Check.Numbers listed on your terminal in numerical order.

"But on a terminal everything goes by so fast that I can't read it," you say. OK, try this!

OS9: list Check.Numbers ! sort ! Who_To ! Tee /P Feb ruary_Checks

Your report should appear on your terminal screen again. But this time your printer will print the same report. OS-9 will also write a copy of the report to a file named "February_Checks". If you need to refer to it in the future, you will find this file in your current data directory. The extra copies of the report were made possible by the OS-9 "Tee" utility command.

When OS-9 executes the above command line it will be running four processes at the same time. Essentially, four simple programs, which were not specifically written to work together, have been combined to do a complex task. Yet, you were able to do a new job without writing a new program.

The OS-9 Shell lets you connect any number of commands with pipes. The new command line is called a pipeline.

SUMMARY

In this chapter you have learned how OS-9 programs communicate with each other and the outside world. We've also introduced you to standard input and standard output paths and have shown you how they can be redirected. Pipes and pipelines gave us a fitting close.

In Chapter 7 we'll introduce you to the OS-9 System Disk and show you how to install OS-9 on your computer. Get a good rest tonight! It's almost time to get your "Hands On" OS-9.

CHAPTER 7

the rainbow os-9 tour guide

Congratulations! You made it through the first six chapters. Aren't you glad you resisted the temptation to turn your computer on before you were ready?

Go ahead, turn it on now. You deserve it. You have built a strong foundation that will help you understand the reason certain OS-9 commands work the way they do. Before long you will have developed a complete repertoire of OS-9 skills.

In this chapter we'll take a quick tour of the OS-9 system disk and explain the various files and directories you'll find on it. Then, we'll bring your computer to life as we show you how to "Boot" the OS-9 system.

Before we're through, you will be able to format a new disk and back up your precious system disk. In fact, you will have made an insurance copy which you can use in an emergency.

INSTALLING YOUR SYSTEM

It's time to go to work. Turn on your computer, grab your Radio Shack OS-9 disks and hold on.

The first thing you need to do is "boot" or load OS-9 into your system. Radio Shack has given you two ways to do this. You must use the first way if you have Disk Extended BASIC Version 1.0. It requires two disks. If you have Disk Extended BASIC Version 1.1, or higher, you only need one disk.

USING ROM VERSION 1.0

If you have Disk Extended BASIC Version 1.0 follow these steps.

1. Turn on your Color Computer
2. Put the disk named OS-9 BOOT in drive 0.
3. Type: RUN "*" <ENTER>
4. You should see this prompt:

   b BOOT OS9
   t TEST DISK DRIVE

5. Type the letter "B" to Boot OS-9.
6. You should see this prompt:

   INSERT OS9 DISKETTE
   INTO DRIVE 0 AND PRESS A KEY

7. Take out the disk marked OS-9 BOOT.
8. Put the disk named OS-9 System Master in drive 0.
9. Strike any key.
10. OS-9 should come alive and ask you for the time.

USING ROM VERSION 1.1

If you have Disk Extended BASIC Version 1.1 or higher, take the following steps.

1. Put the disk named OS-9 System Master in Drive 0.
2. Type: DOS <ENTER>

OS-9 should come to life.

ENTERING THE DATE AND TIME

When OS-9 comes alive, it immediately runs the procedure file named, startup. This file, as shipped by Radio Shack, merely runs the SETIME utility command which starts your system clock and sets the time of day.

You should see a prompt that looks like this:

```
          YY/MM/DD HH:MM:SS
TIME ?
```

Answer this prompt by typing the date and time. You must enter the date and time in one of the the formats shown here:

```
          YY/MM/DD HH:MM:SS
TIME ? 84/08/29 14:34:20

          YY/MM/DD HH:MM:SS
```

TIME ? 84 08 29 14 35 10

Soon after you type the date and time you will see the OS-9 prompt:

OS9:

Congratulations! You are underway.

CAUTION: When you get ready to turn off your computer for the day, make sure that you first take all of the disks out of the drives.

A LOOK AT THE SYSTEM DISK

Almost all OS-9 disk-based computers use a system disk. This special disk holds several directories and files that you will need each time you start OS-9 on your Color Computer. Additionally, many people use their system disk to hold many of the files they use each day.

You will want to leave your system disk loaded at all times. And, since OS-9 usually boots itself from drive number zero, you should leave this disk mounted in drive /D0.

Let's start our tour of your system disk by looking at a listing of the root directory of the Radio Shack OS-9 system disk.

```
        directory of /rs2  11:23:50

OS9Boot      CMDS       SYS       DEFS
startup
```

Looking at this directory, you'll notice two files and three directories. We'll discuss the files first.

If you try to start OS-9 with a disk that does not contain a file named, OS9Boot, it will not work. OS9Boot is a special file because it has been linked. This means that the OS-9 bootstrap program knows how to find it on your disk. To make your own OS9Boot file you must use one of two utilities that come with OS-9.

A command named "cobbler" will put an exact copy of the OS9Boot file you used when you last booted your computer on your system disk. Another, named OS9Gen, lets you pick and choose the modules you store in this file. Both of these utilities insure that your new OS9Boot file is linked. They store it in the root directory of your system disk.

OS9Boot contains all of the modules needed to run OS-9 on a standard 64K Radio Shack Color Computer. We'll show you how to personalize your OS9Boot in Chapter 16. For now, we will work

with plain vanilla Radio Shack OS-9.

In order to show you which modules are stored in the file, OS9Boot, we ran the short form of the OS-9 ident utility command. You can look at it on your own Color Computer screen by typing:

OS9: ident -s /d0/os9boot <ENTER>

```
 2 $E1 $524CEB . CCDisk
82 $F1 $EDF046 . D0
82 $F1 $69933D . D1
82 $F1 $6536D3 . D2
82 $F1 $E155A8 . D3
 3 $E1 $0A6A0A . CCIO
83 $F1 $3EDF55 . TERM
 4 $C1 $BD0579 . IOMan
 6 $D1 $C06CB6 . RBF
 7 $D1 $04D9E6 . SCF
 5 $C1 $23FB76 . SysGo
 2 $C1 $7255DB . Clock
20 $11 $59ECC8 . Shell
 2 $E1 $8D10C1 . RS232
83 $F1 $7BF6CE . T1
 1 $E1 $316E57 . PRINTER
83 $F1 $8080DF . P
 3 $D1 $5F721D . PipeMan
 2 $E1 $5B2B56 . Piper
80 $F1 06AF    . Pipe
```

We described most of these modules for you in Chapter 3. In the table, the first column gives you the edition number; the second lets you look at the type/language byte; the third contains the CRC of the module; the period tells you that the CRC is correct; and finally, the last column tells you the name of the module. If you see a question mark in the column where the periods appear, the CRC of this module named in that line is bad and OS-9 will not load it.

THE OS-9 STARTUP FILE

The other file in the directory listing above is named "startup." This file is not absolutely essential, but its an ideal tool that can save you many keystrokes.

"Startup" is a procedure file or "Shell Script" that tells OS-9 what you want it to do when you bring it to life. If you plan to use a startup file, you must make sure that you store it in the root directory of your system disk. This means that the OS-9 pathlist that describes it will almost always be /d0/startup.

Let's use the LIST utility command and look at the startup file on your Radio Shack System disk.

OS9: LIST /D0/STARTUP <ENTER>
SETIME </TERM

The Radio Shack startup file contains only one line. It tells OS-9 to run the utility command SETIME. This command starts your Color Computer's clock and sets the time. Since the Color Computer does not contain real time clock hardware, SETIME starts a software program in the module named CLOCK that simulates a real clock.

You should note here that the authors of this startup file have redirected the input of SETIME to the device /TERM which is the keyboard on your Color Computer. If they hadn't redirected the input, OS-9 would have tried to read the time from the startup file. Since time never stops, this obviously wouldn't work.

THE CMDS DIRECTORY

The most important directory on your OS-9 system disk is named CMDS. In fact, OS-9 makes it the current execution directory when you boot your system.

Here's an additional point you should know about. People running OS-9 Level I usually store the Shell command utility in their OS9Boot file. However, some don't. If you fall in this latter category, you must make sure that there is a copy of the Shell stored in your CMDS directory.

Most people use the CMDS directory to hold all of the OS-9 Utility commands as well as most of the application programs they use on their computer. When you start OS-9 a special program named SYSGO tells OS-9 that your current execution directory is named /d0/CMDS.

Later, when other users sign on to use your computer from another terminal, the OS-9 LOGIN utility command assigns /d0/CMDS as their current execution directory also. In fact, /d0/CMDS is almost always shared by all users on a computer.

To see the files that are stored in the CMDS directory on your OS-9 System Disk from Radio Shack, type the following command:

OS9: dir /d0/cmds <ENTER>

directory of /d0/cmds 11:24:13

asm	**attr**	**backup**	**binex**
build	**cmp**	**cobbler**	**copy**
date	**dcheck**	**debug**	**del**
deldir	**dir**	**display**	**dsave**
dump	**echo**	**edit**	**exbin**
format	**free**	**ident**	**link**
list	**load**	**login**	**makdir**

mdir	**merge**	**mfree**	**os9gen**
printerr	**procs**	**pwd**	**pxd**
rename	**save**	**setime**	**shell**
sleep	**tee**	**tmode**	**tsmon**
unlink	**verify**	**xmode**	

THE SYS DIRECTORY

Three important files live in the SYS directory on your system disk. Let's take a look. Type:

OS9: dir /d0/sys

directory of /d0/sys 11:24:42

errmsg password motd

The files stored in /D0/SYS are not needed to boot OS-9. However, they are used by the LOGIN, TSMON and PRINTERR utility commands. This means you will need them if you plan to use a second terminal attached to the RS-232 port on your Color Computer.

The file named password is used by the LOGIN command. The file "motd" holds a message of the day that is sent each time a new user signs on to your system. The message is displayed by the LOGIN command.

The file named "errmsg" contains a list of English language error messages that correspond to the OS-9 error numbers. They are used by the PRINTERR utility command. If this file is not stored in a directory named SYS, on a disk mounted on the disk drive named /d0, PRINTERR will not work.

THE DEFS DIRECTORY

The last directory on your system disk is named DEFS. You'll need it if you plan to do any assembly language programming. Let's take a look.

OS9: dir /d0/defs <ENTER>

directory of /rs2/defs 11:25:02

OS9Defs RBFDefs SCFDefs SysType

The four files in this directory contain assembly language source code that assigns English-like names — mnemonics — to the memory locations and routines you use most often. When you write your own assembly language programs, you can also use these names. To do this, you include the OS-9 assembler "USE" directive in your source code. For example:

* Now we'll read the SCFDefs file
USE /d0/defs/SCFDefs
* Names (symbols) in SCFDefs can now be used

OS9Defs is the main system-wide definition file. RBFDefs contains definitions unique to the RBF file manager, while SCFDEfs contains code unique to the SCF file manager. The SysType file holds definitions unique to the hardware you are using.

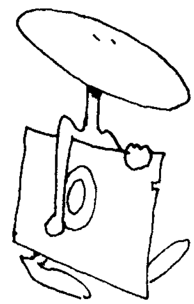
Let's take a look at the file named /D0/DEFS/SysType on your Color Computer:

OS9: list /d0/defs/SysType <ENTER>

Color equ $0A

CPUType set Color
ClocType set 0

DPort set $FF40



FORMATTING A NEW DISK

Before you dive in to any new computer software you should always take the time to make a backup copy of your original disk. Strange things seem to happen to computers and the disks that run in them.

Before you can backup your original Radio Shack system disk, you must format a new OS-9 diskette. Follow these steps:

1. Boot OS-9
2. Type: format /D0 <ENTER>
3. You should see this message:

    COLOR COMPUTER DISK FORMATTER 1.2
    FORMATTING DRIVE /D0
    Y (YES) OR N (NO)
    READY

4. Take out the original Radio Shack System Master disk
5. Put an empty disk in drive 0
6. Type Y
7. You will be asked to name your new disk
8. Type: anyname <ENTER>
9. You will see each track and sector number while Format verifies your new disk
10. When you see the "OS9:" your disk is ready

You should note two things at this point. First, if the OS-9 FORMAT utility finds a bad sector while it is verifying your new disk, it will report it and remove it from the allocation map on the disk. If a sector is not in the allocation map, OS-9 will not use it.

You may go ahead and use a disk with bad sectors for most jobs. However, the OS-9 BACKUP command will not work with a bad disk. This means that you must always use a blank disk that contains only good sectors when you plan to use it with BACKUP.

And second, if you have two disk drives you can leave your system disk in drive /D0 and place your new blank disk in drive /D1. To format the new blank disk, type:

OS9: format /D1 <ENTER>

BACKING UP YOUR SYSTEM DISK

Beginning here we are going to make a very important assumption. Since it is impractical to attempt to do any real work with OS-9 when you only have one disk drive, we are going to assume that you have two drives in your system — /D0 and /D1.

If you are just getting started and only have one drive, you will find the single drive syntax of the OS-9 commands that can be run on one drive in the Radio Shack reference manual named *OS-9 Commands*.

Now that you have formatted a new disk, you may backup your Radio Shack System Master disk by following these steps:

1. Put the disk you just formatted in /D1
2. Put the System Master disk in /D0
3. Type: Backup #20K <ENTER>
4. You will see the following prompt:

   READY TO BACKUP FORM /D0 TO /D1
   ?:

5. Type: Y
6. OS-9 will display this message:

   anyname
   IS BEING SCRATCHED
   OK ?:

7. TYPE: Y
8. Wait patiently while Backup does its job

When the Backup operation is complete, you will see a message on your screen that tells you how many sectors were copied and how many were verified. The two numbers should match.

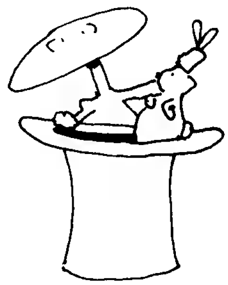
You should now take your original Radio Shack System Master disk and store it in a safe place. From this point on, use only the new system disk you have just made when you work with OS-9.

You're on your way. In this chapter you have been introduced to the OS-9 system disk and the contents of its directories and files. You've also learned how to bring OS-9 to life on your Color Computer — and, how to format new disks and make backup copies of your system software.

Take some time to enjoy your new operating system. Now that you're working with a backup of your original disk, feel free to play around with OS-9. When you're ready to get serious again, join us in Chapter 8 where we'll introduce you to a number of the special keys on your Color Computer keyboard.

CHAPTER 8

special keys



Tricks are for kids! That may be an old saying, but it is quite appropriate here. In this chapter we hope to show you a few tricks that will save your fingers a lot of wear and tear.

We'll be introducing you to two sets of very special keys on the Color Computer keyboard. You'll find yourself using the first set to save a lot of typing. Then, we'll show you how to generate a few of the characters that aren't available on the Color Computer keyboard. You'll need these characters when you start to program with OS-9.

KEYS THAT MAKE LIFE EASY

There are several special key combinations on the Color Computer keyboard that will make your a life a breeze. These keys can help you correct your mistakes or let you run a command line over and over again with only two keystrokes. They can even stop runaway programs.

We're all human and make "misteaks." But, the first time you type LOST in an OS-9 command line when you really mean LIST, you'll discover that OS-9 makes it very easy for you to correct your mistakes. Here's one way to correct an error.

Hit the backspace key several times. Each time you hit the key, you'll notice the cursor backs up one space. Repeat this action until you reach the bad character. Once the cursor is in place, you can retype the character.

On the Color Computer, use the back arrow key to back up

one character position. If you are using a terminal that does not have a backspace key — or a back arrow key — hold down the <CONTROL> key while you type an 'H'. That combination should do the job nicely.

The second way to correct your mistake is to hit the "line delete" key. When you strike this key your computer will erase the entire line and you can start over again. Name your poison.

What did you just mumble? Your keyboard doesn't have a key marked "LINE DELETE."

Don't worry! The Color Computer has an alternative. Just hold down the key marked "CLEAR" and type the letter 'X'. The command line you were typing should disappear. On most terminals you can perform the same magic by holding down the key marked <CONTROL> or <CTRL> while typing the letter 'X'.

If you cannot seem to make the connection between <CLEAR X> and deleting a line, the Color Computer keyboard has another combination that may be easier for you to remember.

You can also delete an entire line with one keystroke by holding down the shift key and striking the back arrow key. The <SHIFT><BACK ARROW> combination to delete an entire line. Got it!

Here is another tip you just may want to remember. When you are using OS-9, the CLEAR key on the Color Computer keyboard always acts like the CONTROL key on other computer terminals.

Other special keys give you a way to repeat your previous input line, interrupt a program, redisplay the present input line, exit a program or simply wait.

The "wait" key does just what its name implies. It stops the text from scrolling on your Color Computer screen until you tell it to start again by striking any key. This gives you a way to stop and study several sentences in the middle of a long text file while you are listing it to your screen.

THE REPEAT KEY

The repeat key will increase your productivity and save your finger tips. You'll love it. Give it a try! Hold down the "CLEAR" key and type the letter 'A'.

You'll find the repeat key is really handy when you need to run the same command line several times. Just type <CLEAR A> and your last command line will magically reappear. Then, type <ENTER> to run the command again.

Here is something you can try the next time you are working

with your Color Computer. Type: dir <ENTER>.

You should see a listing of the contents of your current data directory.

Then type: <CLEAR A><ENTER>. Your trusty Color computer should list the directory again. If you think the repeat key is neat now, wait till you use it with a pathlist 72 characters long.

You'll find that using the <CLEAR A> combination sure beats typing. Use it every time you get the chance.

PUSHING A TASK INTO THE BACKGROUND

If you ever need to interrupt a program while it is running, you can use the OS-9 Interrupt Key. On your Color Computer just hold down the <SHIFT> key and strike the <BREAK> key.

When you type the <SHIFT><BREAK> combination, the Color Computer keyboard sends out a <CONTROL C>. This means that you could get the same result by holding down the <CLEAR> key while you type the letter 'C'.

Here's what happens when you send an interrupt signal to a program. As soon as you type the <SHIFT><BREAK> or <CLEAR C> combination, the OS-9 prompt will appear on your Color Computer screen. But, that's only half the magic. Give it a try. Type:

OS9:list filename >/p

As soon as the printer starts running, type the <SHIFT> <BREAK> combination. Watch what happens.

Did the "OS9:" prompt reappear on the screen? Isn't something strange going on? Why is your printer still printing? What's going on?

Would you believe that when you typed <SHIFT><BREAK>, you told OS-9 to run the printing job as a background task. That's what happened.

To prove it type the list command again. This time leave off the ">/p". Your Color Computer screen should fill with the same listing that is being printed. The printer should continue to print until it finishes the job.

THE I QUIT KEY

When you get tired of a program and want to abort the process, never fret. OS-9 gives you a way to do it. Just type <BREAK>.

You can also stop a program by holding down the <CLEAR> key while you type an 'E'. I guess 'E' stands for "End it!"

Here's an historic sidelight. On early OS-9 computer systems you typed <CONTROL Q>. 'Q' for quit. Got it?

That combination was easy to remember. Then progress got in the way. When OS-9 Level II was released it supported the X - ON / X - OFF protocol. Since the ASCII X - ON code is a <CONTROL Q> there was a natural conflict. So much for easy mnemonics.

Here's another handy key. Sometimes you need to redisplay the command line you are typing. To do this type <CLEAR D>. 'D' for display, maybe?

THE GREAT ESCAPE

OS-9 has one more special key. It lets you ESCAPE. The <CLEAR><BREAK> combination sends an end-of-file signal to OS-9. This gives you a way to send an end of file signal to any process that receives its data from the keyboard.

How do you send it? I bet you can't guess. Would you believe that you hold down the <CLEAR> key and strike the <BREAK> key.

There's only one catch to the great <ESCAPE>. When you send the ESCAPE code to OS-9, you must type it as the first character on the line.

THE <CLEAR><NOTHING> KEY

Oops! I almost forgot something — I mean <CLEAR><NOTHING>. The <CLEAR><0> key combination lets you toggle the shift lock on the keyboard. If your keyboard is only sending out uppercase letters, you can get it to send lowercase letters by holding down the <CLEAR> key and typing a '0'.

To change back you simply type the same combination again. That's why we call it a toggle. By the way, when the keyboard is sending out lowercase letters, you can demand an uppercase letter by holding the <SHIFT> key.

Here's an interesting problem to ponder. It is possible to type lowercase letters on the keyboard but only see uppercase letters on the screen. Why?

This happens when you set the TMODE uppercase lock mode to UPC. To see the lowercase letters again, use this command line:

OS9: tmode -upc <ENTER>

Remember, the shift lock function — the <CLEAR><0> key combination — only works when you have used the TMODE utility command to tell the /TERM device descriptor to recognize both upper and lowercase characters.

OTHER OS-9 MAGIC

Are you impatient? Do you hate to sit and wait for the computer to finish one job so you can command it to do another? Wait no more! OS-9 lets you "type ahead."

While OS-9 is running one program, you can type another command line, or answer the next prompt if you know what it is going to be. Sometimes you may be able to stay several command lines ahead of your Color Computer.

Unfortunately, there are two "gotchas" with type ahead on the Color Computer. First, you will be typing blind. This is only a minor slow down and is much better than sitting around twiddling your thumbs. Secondly, you will find that you cannot type ahead reliably on the Color Computer keyboard while the disk drives are being used.

GENERATING ADDITIONAL CHARACTERS

Since there are only 50 keys on the Color Computer keyboard, Radio Shack had to come up with a way to generate a number of characters needed by the high level languages that run under OS-9. The keys that generate these characters are a standard item on most computer terminals.

The table below shows the character generated, its name, the key combination required to generate it and a description of its appearance.

KEY	NAME	KEY COMBINATION	APPEARANCE
_	Underline	<CLEAR><->	Back Arrow
{	Left Brace	<CLEAR><,>	Left Bracket (REVERSE VIDEO)
}	Right Brace	<CLEAR><.>	Right Bracket (REVERSE VIDEO)
~	Tilde	<CLEAR><#>	Hyphen (REVERSE VIDEO)
\	Backslash	<CLEAR></>	Slash (REVERSE VIDEO)
|	Vertical Bar	<CLEAR><1>	Exclamation (REVERSE VIDEO)
^	Up Arrow	<CLEAR><7>	Up Arrow
[	Left Bracket	<CLEAR><8>	Left Bracket
]	Right Bracket	<CLEAR><9>	Right Bracket

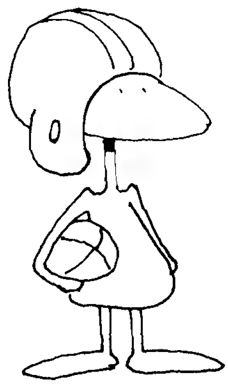
SUMMARY

Special keys make OS-9 easy to use. In this chapter, you've been introduced to keys that save typing and keys that generate characters unavailable on the Color Computer keyboard.

In Chapter 9 we let you get a feel for the OS-9 Shell as we introduce a handful of commands. It will give you a chance to get ready for our six chapter tour of the complete OS-9 utility command set.

CHAPTER 9

a little practice



This chapter is a warm up exercise designed to give you a chance to shake your stage fright. You can't talk about computing forever. You need to start computing. The short tour in this chapter should make you feel confident. Before we finish you'll be able to:

Make a directory
Build a file
List a file
Use a file
Change a file

Before a baby can walk, it must learn to crawl. Before a football team can win a game, each player must learn to block and tackle.

You are a programmer. OS-9 is a programming tool. Together, you are a team. But, as with any team, you must learn the basics and practice them before you can turn pro.

Each utility command in your OS-9 CMDS directory is a tool. Most of these tools work on files. But let's take first things first.

Before you can work on a file, that file must exist. You must create it. BUILD is a handy utility command that makes it easy for you to create small files on a disk so you can use them later.

These files may contain simple messages that OS-9 can list to your screen later when you need a reminder. Or, they may contain a list of commands to put your Color Computer through its paces automatically. A file that does the latter is called a procedure file.

Essentially, a procedure file is nothing more than a sequence of short OS-9 command lines that do a big job when they are run together. An OS-9 procedure file is similar to a shell script on a UNIX computer.

For larger files you will need to use an editor to enter your text or procedures. EDIT, which comes on your OS-9 system disk, is an excellent line editor. If you prefer a screen oriented editor, we suggest DynaStar.

CREATING A DIRECTORY

In Chapter 5 we introduced you to hierarchial directories and showed how they can help you organize your information. In a way, these directories are like the files we talked about above. Before you can use them, you must create them. In fact, that is our first assignment.

Since we plan to keep all our files short during this chapter, you can go ahead and use the backup of the Radio Shack system master disk you created in Chapter 7. There should be plenty of room on it for a few short files.

After you boot OS-9, your current data directory is usually the root directory of device /D0. The execution directory is usually /D0/CMDS.

Let's start by creating a directory you can use to store your files. Try this command line:

OS9: makdir TEST_DIRECTORY

Notice that we typed the name of the directory in all capital letters. Upper case letters make directories easy to spot in a directory listing.

Also notice that we didn't need to type:

OS9: makdir /D0/TEST_DIRECTORY

Since our current data directory was /D0, OS-9 was smart enough to put our new directory in the root directory of device /D0.

Now that we have a directory for you to use, let's move you into it. Type:

OS9: chd test_directory

Notice that you didn't need to type the directory name in capital letters when you called for it. The same goes for filenames. OS-9 will match either upper- or lowercase letters.

BUILDING A FILE

Let's create a file. Type:

OS9: build my_first_file <ENTER>

OS-9 will print a question mark on the screen and wait for you to type a line.

OS9: build my_first_file <ENTER>
? This is my first file. <ENTER>
? I think I'll add a second line to it. <ENTER>
? <ENTER>

After you type the ENTER following the third question mark, OS-9 will save your file on the disk in drive /DO. It will be stored in the directory named TEST_DIRECTORY.

Putting a message in a file can be as simple as the example above. Next, we'll attempt to prove that you did, indeed, create a file and it *was saved* on the disk. Type:

OS9: dir <ENTER>

OS-9 should reply:

DIRECTORY OF . 12:35:45

my_first_file

Congratulations! You have now created a directory and a file. Do you remember what's in it?

LISTING A FILE

You can read the data stored in any text file by listing it to your terminal. You do this with the OS-9 list utility command. Go ahead and try it. Type:

OS9: list my_first_file

OS-9 should oblige:

This is my first file.
I think I'll add a second line to it.

"My_first_file" is an example of a message that can be printed on your terminal. You can create other files that do things. We call them procedure files. Let's make one. Type:

OS9: build greetings
? echo Good morning Dale L. Puckett

? echo It's about time you showed up
? echo The correct time is:
? date t
? echo I've checked your files
? echo Here's a list of your directory
? dir
? echo Good Bye
? <ENTER>

What do you think will happen when you run this procedure file? Let's give it a try.

USING A FILE

Procedure files are usually stored in the current data directory. Let's see how OS-9 handles the procedure file you just created. Type:

OS9: greetings <ENTER>

When you type this command line, OS-9 looks for a module named greetings in its module directory. Most likely, it will not find it.

Then, it looks for a file named greetings in the current execution directory. The odds are very good that it won't find it there, either.

Finally, OS-9 looks for a procedure file in the current data directory. Since you just saved "greetings" in this directory, OS-9 should find it there and execute each command line in it. You should see something like this on your screen.

OS9: greetings <ENTER>
Good Morning Dale L. Puckett
It's about time you showed up
The correct time is:
August 29, 1984 23:33:55
I've checked your files
Here is a listing of your directory
DIRECTORY OF . 23:34:05
my_first_file
Good Bye!
OS9:

If you would like a greeting like this each time you start OS-9 on your Color Computer, BUILD a file named greetings in the root directory of your system disk /d0 and put the word "greetings" in the file /d0/startup. Your new "startup" file would look like this:

SETIME </TERM
greetings

At other times you can use the BUILD utility command to save information. For example, a short list of names and addresses is a very helpful thing to have handy on your computer. It sure beats searching through several hundred business cards.

Here's how you might build a list of names and numbers. Type:

OS9: build address_list
? Rainbow, Prospect, KY 40059
? Puckett, Dale L.; Dale City, VA 22193
? Pollution Response Branch, USCG Headquarters 20593
? <ENTER>

As long as your list of names and addresses is short, you can use the OS-9 list utility command to find a name. Later, when the list grows you can use one of the more powerful OS-9 pattern matching utility commands — GREP from Microware's OS-9 Toolkit, for example — to find a single entry in your file.

CHANGING A FILE

What happens when someone in your name and address file moves? You'll need to find a way to change the address. You can do this with the OS-9 EDIT utility command.

Edit is an extremely powerful text editor that you can use to prepare and edit text files. You can use its macro capability to automate many tasks. Here, we'll only show you a few of the basics so you can use it to enter and edit a file.

Here's how you can edit the address file you created with the build utility command above. Type:

OS9: edit address_list

The OS-9 Edit Utility Command will load, and in a few seconds your screen should look like this.

OS9: edit address_list

E:

The "E:" is a prompt that tells you that Edit is waiting for you to give it a command. Let's start by making sure we have the right file. To list the entire file, type:

E: l* **<ENTER>**
Rainbow, Prospect, KY 40059
Puckett, Dale L.; Dale City, VA 22193
Pollution Response Branch, USCG Headquarters 20593

Now let's insert a new name at the beginning of the file. Type:

E: <SPACEBAR> Dundon, Dick; Kent, WA 98042 <ENTER>

Let's see if it is in place. Type:

E: -*l* <ENTER>

You should see:

E: -*l* <ENTER>
Dundon, Dick; Kent, WA 98042
Rainbow, Prospect, KY 40059
Puckett, Dale L.; Dale City, VA 22193
Pollution Response Branch, USCG Headquarters 20593

Good! Now, let's put a new name and address at the bottom of the file. Type:

E: / <ENTER>
E: <SPACEBAR> Hogg, Frank; Syracuse, NY 13202 <ENTER>

Now let's see if it is in the right place. Type:

E: -*l* <ENTER>

You should see:

E: -*l* <ENTER>
Dundon, Dick; Kent, WA 98042
Rainbow, Prospect, KY 40059
Puckett, Dale L.; Dale City, VA 22193
Pollution Response Branch, USCG Headquarters 20593
Hogg, Frank; Syracuse, NY 13202 <ENTER>

Now let's imagine that Dick Dundon moves to the Silicon Valley. We'll need to change his address. Type:

E: -* <ENTER>
E: C/Kent, WA 98042/Sunnyvale, CA 94087/ <ENTER>

Now check your file.

E: -*l* <ENTER>
Dundon, Dick; Sunnyvale, CA 94087
Rainbow, Prospect, KY 40059
Puckett, Dale L.; Dale City, VA 22193
Pollution Response Branch, USCG Headquarters 20593
Hogg, Frank; Syracuse, NY 13202 <ENTER>

The sample editing session above should give you a feel for the OS-9 Edit utility command. Here is a table that gives you a handful of editing commands to help you get started. After you master these, study the the sample sessions in the Radio Shack

"OS-9 Program Development" manual. You'll have the edit command aced in no time.

COMMAND	ACTION
<SPACEBAR>	Inserts text following the <SPACEBAR>at the position of the edit pointer
<ENTER>	Moves edit pointer forward one line
+	Moves edit pointer forward one line
+6	Moves edit pointer forward six lines
+*	Moves edit pointer to bottom of file
/	Moves edit pointer to bottom of file
-	Moves edit pointer back one line
-4	Moves edit pointer back four lines
-*	Moves edit pointer to top of file

C/old string/new string/
Changes first occurrence of "old string" to "new string"

C3/old /new /
Changes next three occurrences of "old" to "new"

C*/bad word/good word/
Changes all occurrences of "bad word" to "good word"

Edit has many other commands that can make your editing simple. After you master these, dig in.

There's one thing you should remember. When you give one of the commands above to Edit, you must start typing it at the first character position in the line. If you add any <SPACES>, Edit will insert a line for you.

And finally, when you are satisfied with your data file and are ready to stop editing, don't forget to type:

E: Q <ENTER>

This will cause Edit to save your file in your current data directory and return you to the Shell.

SUMMARY

We've eased you into OS-9 gently in this chapter. As you can see, if you followed us along on your computer, OS-9 can be a lot of fun. Stick with us though. We all have a lot to learn. In the next chapter we start a six chapter tour of the complete OS-9 utility command set. We begin by showing you commands that give you information.

CHAPTER 10

commands that give you information

Man invented the computer to help him manage information. Once he had a computer, he learned to store data in files that contained either text or numbers. He then wrote application programs to manipulate this data and display it for him.

In fact, you probably use more application programs on your computer than anything else. However, when you get bored just using your computer and want to start programming, you'll find that you'll need a lot of additional information about your operating system. You'll want to know what it is doing. The commands in this chapter display reports that give you this information.

In this chapter you'll be introduced to the following OS-9 utility commands.

date
display
echo
free
ident
mdir
mfree
printerr
procs

We'll start with a simple command — date.

DATE

There's really not a whole lot you can say about the OS-9 Date utility command. It does just what its name implies. It displays the month, day and year on your Color Computer screen.

You may also ask DATE for the time by including the letter ' t ' on your command line. Let's try it both ways. First, type:

OS9: date <ENTER>

Since the date utility writes its output to the standard output path, you should see something like this on your screen.

OS9: date <ENTER>
September 23, 1984
OS9:

You can also send DATE's output to any hardware device attached to your Color Computer or to a disk file. For example, try:

OS9: date t >/p <ENTER>

In a few seconds, your printer should come alive and print a line that looks like this:

September 23, 1984 20:38:05

The OS-9 date utility, unlike its UNIX equivalent, does not let you set the date. To set the time on your Color Computer, you must use the OS-9 Setime utility command described later.

DISPLAY

When you want to look at straight English language text files, you can display them on your Color Computer screen or another terminal with the OS-9 List utility command. You can also send English language messages to your screen or any device attached to your computer with the OS-9 Echo utility command.

However, life is not always that simple. Sometimes you need to send a character to your screen that is not in the English alphabet. An example of such a character is the so called control code that you use to clear the screen on your Color Computer screen. Also, you often need to send a control code or two to your printer to make it do something special, like issue a form feed or underline a word.

You'll find a complete list of special control codes in Appendix B of the Radio Shack OS-9 Commands manual that will make your Color Computer dance. This appendix also gives you codes you can use to display graphics from an OS-9 command line.

To send these special characters — or control codes — to your Color Computer screen or printer, use the OS-9 DISPLAY utility command.

To use DISPLAY, just type the word "display" followed by one or more hexadecimal numbers. The DISPLAY utility converts the hexadecimal number you type to an ASCII character and writes it to the standard output path.

Let's try it!

OS9: display C <ENTER>

This command line will home the cursor and clear the screen on your Color Computer. The 'C' in this command line is the hexadecimal equivalent of the decimal number 12 listed in Appendix B of the Radio Shack OS-9 Commands manual. Remember, you must always type hexadecimal numbers when you use display.

Here's another example:

OS9: display F >/p <ENTER>

This command will cause an Epson MX-80 printer to start printing compressed text.

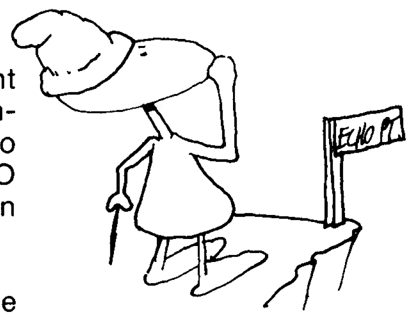
And finally:

OS9: display 31 32 33 34 35 36 37 38 39 30 <ENTER>
1234567890

This DISPLAY shows you another — albeit harder — way to count to 10.

ECHO

Quite often you need to remind yourself to push the right button. You can do this by using the Echo utility command in your Shell procedure files.



Actually, you can use the OS-9 ECHO utility command to print any message written in English language text on your Color Computer screen at any time. You can also redirect ECHO's output to another terminal or your printer. This means you could use ECHO to send a message to someone working on a second terminal in another room.

When using ECHO, you should make sure that you don't type any of the special characters recognized by the Shell. The results may not amuse you.

Let us ECHO.

OS9: echo >/t1 I'm starved, let's have dinner.<ENTER>

This command line will display the message, "I'm starved, let's have dinner." on the terminal device named /t1.

Here's another!

echo >/term ** ATTENTION ** Formatting Disk!

You could put this command line in a procedure file and have OS-9 remind you when it is getting ready to format a disk. We often need that one last chance to change our minds.

Since the OS-9 Shell does not process meta-characters, the ECHO utility is not as powerful as its UNIX equivalent. The UNIX ECHO lets you substitute commands, variables and all meta-characters recognized by the Shell.

A meta-character is any character that has a particular special use other than its normal use within the alphabet. For example, the OS-9 Shell recognizes the exclamation point as a signal to set up a pipe. The exclamation point is a meta-character.

FREE

Sometimes you need to know how much space you have left on a disk. The OS-9 Free utility command does this job nicely. To find out how many sectors of free space are left on a disk, just type the word FREE followed by the name of a disk drive. You can even tell how many new files can be stored on a disk by studying the information provided by the FREE utility.

As a bonus, you will also learn when the disk was created and the size of its clusters. The number of sectors in a cluster is important because it has an effect on the number of files that you may store on a disk.

Imagine that you are using disk drives that have 16 sectors in a cluster and FREE tells you that 48 sectors are available. You will know that you can only create three new files. This limit exists because OS-9 can only read or write an entire cluster on a disk at a time.

Let's give it a try! Type:

OS9: free /d2 <ENTER>
DynaSpell_Documentation created on: 84/06/26
Capacity: 1,274 sectors (1-sector clusters)
1,070 free sectors, largest block 940 sectors

This command line caused free to report that a disk named "DynaSpell_Documentation" was mounted in device /d2. Further it reported that the disk was created on June 26, 1984 and that it

has a capacity of 1,274 sectors. Of those 1,274 sectors, 1,070 remain free and may be used to store your data.

IDENT

OS-9 is a modular operating system. This means that it is not one long program. Rather, it contains a number of small modules that work together.

You can find the names of modules present in memory with the MDIR utility command. Sometimes, however, you need to know more about a module. That's where the IDENT utility comes in.

When you run IDENT, it reads the header of the module named in your command line and displays information from it on your Color Computer screen. For example, it tells you the size of the module and prints the value of its CRC. It also tells you if the CRC is good or bad.

If a module contains object code that will execute, IDENT tells you where the actual program code starts by displaying the offset from the beginning of the module. It also tells you how much memory the program needs in order to run.

The Ident utility tells you what type of data is contained in a module and the language that uses it. Further, it displays the revision number of a module and its attributes. When you run IDENT on a disk file, it displays a report about each module stored in the file.

You can use four command line options with IDENT: -m, -s, -v and -x.

The -m option tells IDENT to look for the module in memory.

The -s option causes IDENT to print a short report.

If you type a -v in your command line, IDENT will not verify the CRC of the module you are checking.

And finally, if you type a -x, IDENT assumes that the file named is stored in your current execution directory.

Here's a trial run!

OS9: ident -m spell

```
Header for:   SPELL
Module size:  $25EE    #9710
Module CRC:   $AE36FF  (Good)
Hdr parity:   $E3
Exec. off:    $0DCF    #3535
```

```
Data Size:    $4BDC   #19420
Edition:      $2F     #47
Ty/La At/Rv:  $11 $81
Prog mod, 6809 obj, re-en, R/O
```

In this example, IDENT read the header of a module named spell. Since we typed the -m option, it looked for the module in memory rather than a file. Notice that it verified the modules CRC and reported that it was good. Can you tell the difference between this report and the next.

```
OS9: ident -mv spell

Header for:   SPELL
Module size:  $25EE   #9710
Module CRC:   $AE36FF
Hdr parity:   $E3
Exec. off:    $0DCF   #3535
Data Size:    $4BDC   #19420
Edition:      $2F     #47
Ty/La At/Rv:  $11 $81
Prog mod, 6809 obj, re-en, R/O
```

Notice that the "(Good)" is missing from the report. IDENT did not verify the CRC of the module spell because we used the -v option. Now, let's ask for a short report of a file stored in the current execution directory.

```
OS9: ident -xs ds

30 $11 $A3C080 . DS
 3 $21 $749D97 . PINTERP
```

Notice that when you run the short form of the IDENT utility command, a period shows you that a module's CRC is correct. If the CRC is bad, the short form of IDENT will print a question mark in the same column.

MDIR

Since OS-9 contains a number of modules, it is often handy to know which modules are in memory. You can get a listing of the names of each module in memory by running the MDIR utility command.

If you need more information, you can use the extended form of the MDIR command. To do this, add the letter 'e' to your command line. The extended module gives you the names of all modules in memory and shows you where each module is loaded, how many bytes it contains, the type of code it contains, its revision number and the number of processes presently using it.

When you run MDIR on a Level II system, you will also learn

the extended physical address of a module. Let's run MDIR now and see what happens.

OS9: mdir <ENTER>

Module Directory at 16:40:23

Boot	OS9p1	OS9p2	Init	SysGo
IOMan	RBF	SCF	ACIA	PIA
PipeMan	Piper	Pipe	Clock	TERM
T1	T2	P	P1	d0
d1	d2	d3	M1	ESTERM
DS	PINTERP	SPELL	Lk	DF

That was pretty cut and dried. Now, lets see what kind of information we can find by using the extended form of the MDIR command.

OS9: mdir e <ENTER>

Module Directory at 16:47:39

Block	Offset	Size	Typ	Rev	Attr	Use	Module Name
FF	0	2B0	C1	2	r...	1	Boot
FF	2B0	D26	C0	8	r...	0	OS9p1
1	0	C64	C0	2	r...	1	OS9p2
1	C64	2E	C0	1	r...	1	Init
1	C92	75	11	1	....	1	SysGo
1	D07	929	C1	1	r...	1	IOMan
1	1630	10D5	D1	1	r...	4	RBF
1	2705	4C2	D1	1	r...	5	SCF
1	2BC7	2A1	E1	1	r...	2	ACIA ...

MFREE

When you run a powerful operating system like OS-9 on a small microcomputer like the Color Computer, you soon learn that you have limited resources. In fact, on any OS-9 Level I system, memory is probably the most precious resource.

To manage your memory wisely you need to know how much you have available. The MFREE system utility command answers some of your memory questions.

When you run MFREE, it prints a list of the memory areas in your computer that are not being used. If they are not being used, they are available for you to use.

Additionally, MFREE gives you the address where each free block starts and reports its size. On a Level I system like the Color Computer, MFREE tells you how many 256-byte pages are available.

MFREE also shows the block number, physical beginning and ending addresses, and the size of each memory area on Level II systems. The Level II size is reported as both the number of blocks and the number of free (K)ilobytes available.

Here's how you run MFREE.

OS9: mfree <ENTER>

Address	pages
800- 8FF	1
B00-AEFF	164
B100-B1FF	1

Total pages free = 166

The command line above was typed into Color Computer OS-9. You should see a similar report on any Level I OS-9 system.

OS9: mfree <ENTER>

Blk	Begin	End	Blks	Size
70	70000	77FFF	8	32k
80	80000	87FFF	8	32k
90	90000	97FFF	8	32k
		Total:	18	96k

When you type mfree on a Level II OS-9 system, you'll see a wider listing. If you study the report above, you will notice that MFREE tells you which block of memory is free and where the free memory starts within the block.

Remember, the memory areas and amounts displayed by the MFREE utility command are not in use. This means they are available for you to use. Of course, since OS-9 programs are all position independent, you will be concerned mainly with the amount of memory available. MFREE can help you track down the source of a memory fragmentation problem, however.

PRINTERR

Some computers have a very bad habit. When you make a mistake, they are unforgiving. They tell you that you blew it, but they don't tell you how. If you're lucky, they throw an error number at you. Some help!

OS-9 has a utility command that helps solve this problem. Once you run it, the system will report your errors in English. Believe me, it helps a lot.

Printerr displays English language error messages from a file named /d0/SYS/errmsg. Once you run PRINTERR, it replaces the standard OS-9 error reporting routine that only prints error code numbers.

Printerr installs itself permanently the first time you run it. There's one thing you should know, though. Once you run PRINTERR, you are stuck with it for the rest of your OS-9 session. It literally attaches itself to your computer. If you need to know how to change the existing error message file or install your own, consult the Radio Shack OS-9 Users Manual.

The Printerr command syntax could not be any simpler. Just type:

OS9: printerr <ENTER>

Let's make a mistake on purpose and compare the reports received both before and after this routine is installed.

BEFORE PRINTERR

OS9: datte <ENTER>
ERROR #216

AFTER INSTALLING PRINTERR

OS9: PrintErr <ENTER>
OS9: datte <ENTER>
ERROR #216
- PATH NAME NOT FOUND

Remember, once you have installed PrintErr you cannot remove it. DO NOT try to unlink PrintErr. You will crash the system. The only way to remove PrintErr once it has been run is to reboot the system.

PROCS

Sometimes you may notice that your terminal seems to be sluggish. Or, a sort may take longer than usual. This happens when your system is loaded down with too many processes.

Also, if you're snoopy, its nice to know who is doing what on the system. OS-9 has a utility command that can give you all of this information. It's called PROCS.

When you run PROCS, it gives you a list of processes that are running on your computer. Normally, it only lists the processes you own. However, you may ask to see all processes. To do this, you just add the 'e' option in your command line

PROCS reports the user number of the owner of each process

and displays each process ID number. The state of the process, its priority, and the amount of memory it is using are also given. The primary program module and the standard input path for each process are also displayed.

The Level II version of PROCS gives you the process ID number for each process running in your computer. It also reports the ID number of the parent process, the priority of a process, the amount of memory being used by the process and the current address of the stack pointer.

Again, the command syntax is simple.

OS9: procs <ENTER>

Here's the result on an OS-9 Level II system.

ID	Parnt ID	User Numbr	Pty	Age	Sts	Signl	Mem Siz	Stack Ptr	Primary Module
2	1	0	128	129	$80	0	1	$97E2	SysGo
3	2	0	128	129	$80	0	3	$95E2	Shell
4	3	0	128	129	$80	0	128	$93E2	DS
5	0	0	128	129	$80	0	2	$90E2	Tsmon
8	4	0	128	128	$80	0	3	$82E2	Shell
9	8	0	128	128	$80	0	6	$05F3	Procs

The report above displays only those processes that are owned by user number 0. To see a listing of all the processes being run on your computer you must add the 'e' option to your command line. Try this:

OS9: procs e <ENTER>>

ID	Parnt ID	User Numbr	Pty	Age	Sts	Signl	Mem Siz	Stack Ptr	Primary Module
2	1	0	128	129	$80	0	1	$97E2	SysGo
3	2	0	128	129	$80	0	3	$95E2	Shell
4	3	0	128	129	$80	0	128	$93E2	DS
5	0	0	128	129	$80	0	2	$90E2	Tsmon
6	5	2	128	128	$80	0	3	$8BE2	Shell
7	6	2	128	128	$80	0	125	$88AE	Stylo
8	4	0	128	128	$80	0	3	$82E2	Shell
9	8	0	128	128	$80	0	6	$05F1	Procs

Notice that there are two extra processes visible in this listing. We can see that a process named Stylo is running. It was started by a Shell that was started by Tsmon. Both the Shell and Stylo are owned by User Number Two. That's why they did not show up in the first PROCS report.

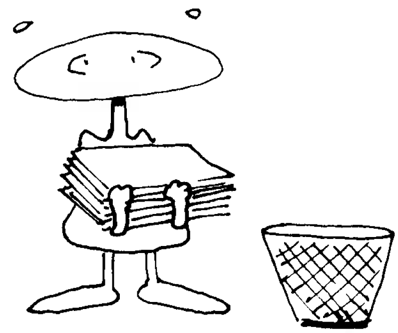
SUMMARY

In this chapter we have introduced you to nine programs from the OS-9 utility command set. Each of these commands tells you something about your system. In Chapter 11, we look at the OS-9 commands needed to work with files.

CHAPTER 11

commands that work on files

OS-9 is a disk based operating system. That means that it stores most of its own information in disk files. Likewise, when you run an application program you will be storing a lot of information in disk files.



In this chapter we'll concentrate on the OS-9 utility commands that help you work with files. You'll be introduced to:

attr	dump
binex	exbin
build	list
cmp	merge
copy	rename
del	tee

ATTR

Each OS-9 disk file has a set of qualities that define certain characteristics about it. Microware and Radio Shack both call these qualities attributes.

For example, some files contain 6809 object code that you load into your computer and run. Since these files can be run — or executed — OS-9 assigns an executable attribute to them.

Other files often contain English language text that can be sent to your screen or printer. But sometimes, the words in these files may be a secret. For example, if the wrong person reads a file containing propriety information that could help a competitor and then sells it, your company stands to lose money.

Because computers are often used to store this type of information, OS-9 lets you give a file a "readable" quality or attribute. With this attribute you can mark a file so that it can be read by everyone running your computer. Or, you can mark it so that you are the only one who can read it.

The name of the attribute that tells OS-9 that a file can be read by everyone on the system is "pr." The letters "pr" stand for public read. To mark a file so that you are the only one who can read it, you use the 'r' or "read permit to owner" attribute.

Now, ponder this. For every "pr" attribute there is an equal and opposite "-pr" attribute. A file that is marked with a -pr can not be read by anyone except the owner. The same story is true for the 'r' attribute. If you don't want anyone to be able to read a file, including yourself, you may tell OS-9 to give the file the "-r" quality or attribute.

As you can see, attributes can be used to protect your files from unauthorized operators — or yourself. But how do you mark these files, or, in the language of the hacker, how do you set their attributes?

OS-9 has a special utility command that lets you look at the attributes of a file. Fortunately, it will also let you change them. The name of this utility command is ATTR.

To use ATTR, you should be familiar with all of the qualities or attributes available to a file. They are r, w, e, pr, pw, pe, d, s. What do you think they mean?

I'll bet you don't have to stretch your imagination too far? However, in the interests of being complete, let's review.

In general, the single letter attributes, r, w and e apply to the owner only. A file that has the 'r' attribute set can be read by the owner. If the 'r' attribute is forced negative, or set to -r, even the owner will not be able to read the file.

The 'e' and 'w' attributes work the same way. A file that has the 'e' attribute set can only be executed by the owner. If the 'e' is set to a "-e," even the owner will not be able to execute the file. Pursuing the same logic path, if a file has the 'w' attribute set, the owner can write to it. If the file has a "-w" attribute, not even the owner can write on it.

The two letter commands deal with public access. If a file has the "pr" attribute set, anyone on the system can read it. If the "pr" attribute is forced negative or set to "-pr", and the 'r' attribute is set, only the owner of the file will be able to read it.

The "pw" stands for "public write. The "pe" stands for public execute. Both work just like the "pr" attribute above except that

they affect the public's ability to write or execute a file rather than the ability to read a file.

The rest of the file attributes available on an OS-9 system are easy to explain. For example, a file that has the 'd' attribute set, contains a directory.

The 's' attribute stands for "sharable." When it is set, the file may only be used by one person at a time. It becomes a "single user file" for want of a better name. For example, if you write a program that uses non-reentrant code you would need to set the 's' attribute on the file that holds it.

In addition to the file attributes described above, you may use one option, "-a" on an ATTR command line. This option tells OS-9 not to print the file's attributes after they are changed.

Let's take a look at ATTR in operation. Type:

OS9: attr KISSable_OS9 -pr -pw r w <ENTER>
---wr---

Our command line cleared the public read and public write attributes of a file named, KISSableOS9. At the same time, it set the 'r' and 'w' attributes on that file. This means that only the owner —the person who created the file — can read and write to it.

Actually, that's not quite true. If you are responsible for the security of your computer system, you should know that user number 0 on an OS-9 based computer is called the "superuser." The superuser can read and write to any file on the system.

After ATTR changed the attributes of our file, it reported its action in the next line. Here's what that mysterious looking line means.

If there is a dash or a negative sign in a position, that attribute is negative. A request for an action that requires that attribute to be set will be denied. If a letter appears in a position, that attribute is set and the corresponding action would be allowed.

An ATTR report with all attributes set would look like this:

dsewrewr

If all attributes were clear, or negative, an ATTR report would display the following line.

The attributes flags in the two report lines above appear in the following order:

d, s, pe, pw, pr, e, w, r

Or:

directory
sharable
public can execute
public can write
public can read
only owner can execute
only owner can write
only owner can read

Here's another example:

OS9: attr important_data -w -pw -e -pe -a <ENTER>

This command lines shows how you can write protect a file. When both the public, "pw," and private, 'w,' write attributes are clear, no one can write to that file — not even the system.

Since it is impossible to write to the file, it is also impossible to delete or rename it. Notice that since you used the "-a" option, ATTR did not echo a report after it made your changes.

After you clear the write and public write attributes of a file you own, you are the only one who can delete the file. In fact, it will even be a hassle for you to delete the file.

To get the job done you will need to use the ATTR utility command to reset the write attribute. After you do this you are free to delete the file. But since you own the file, you are the only person who can use the ATTR utility command on the file.

Oh, by the way! If you're thinking about playing games with another person's file attributes, don't bother. You cannot change the attributes of a file you don't own.

Here's another interesting fact. You can change a directory to a file if there are no files in it. In fact, that's how the deldir utility works.

BINEX AND EXBIN

After you develop an excellent piece of software, you usually want to share the wealth — or at least try to find it. This often means that you must transfer binary object code from one computer to another. You do this by porting — or sending — the object code through an RS-232 port directly. You can also transmit your code over a telephone line connected to your computer with a modem.

One of the most common ways to port binary code is to send it in the Motorola S-record format. The Motorola scheme converts the binary object code into a series of hexadecimal characters that you can transmit in ASCII. It also provides a CRC check to ensure that each record is received correctly.

BINEX reads a file stored in OS-9's format, converts it into the Motorola format, and stores it in another file. The new file can then be transmitted to another computer.

EXBIN does just the opposite. It reads a file that has been written in the Motorola format, converts it, and then writes it to a file in the OS-9 binary format.

Here's how you call BINEX from the OS-9 Shell:

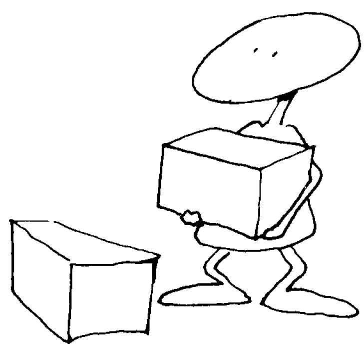
OS9: binex /d0/cmds/dir dir.Motorola <ENTER>
Enter starting address for file: 0 <ENTER>
Enter name for header record: dir <ENTER>

Notice that OS-9 prompted you for the loading address of the binary code and asked you for a name to put in the Motorola file's header record. It does this because the Motorola S1 format requires this information.

A starting address is meaningless to OS-9 because all OS-9 modules must be position independent. In fact, all OS-9 assembler code is written so that its starting address is zero.

After running the command line above, you will find a file named "dir.Motorola" in your current data directory. You can send this new file to another computer connected to terminal device /T1 like this:

OS9: list dir.Motorola >/T1 <ENTER>



The person you are sending this file to will need to store it temporarily in a file. After doing this he can run the EXBIN command to convert it back so that he can run it on his own OS-9 based computer. Here's the command line:

OS9: exbin newcat.Motorola /D0/CMDS/newcat

BUILD

Since we let you exercise the BUILD utility command quite a bit in Chapter 9, we'll try to keep our dialogue short here.

BUILD is an OS-9 utility command that you can use to write short text files. Generally, you type English language messages on your keyboard and OS-9 stores them in a file. Additionally, you can type OS-9 procedure files — or Shell scripts — and have BUILD store them in a file.

Since BUILD reads its input from the standard input path, you may send it input from a file as well as your keyboard. Remember, on most OS-9 based computers, the standard input path is usually connected to the keyboard on your terminal. BUILD sends its output to the file that you name on your command line.

When you run BUILD, it opens a path to the file or device that you name. Then, it sends you a prompt.

The question mark that BUILD displays on your screen means that it is waiting for you to type a line. After you type an empty line by hitting your <ENTER> key two times in a row, BUILD saves the file and returns you to OS-9.

Let's give it a try, just for review. Type:

OS9: build greetings <ENTER>
? Good Morning <ENTER>
? Welcome to OS-9 Dale! <ENTER>
? <ENTER>

The sequence above will cause your computer to create a file named "greetings" in your current data directory. It will contain two lines. To see what is stored in that file you can use the LIST utility command. Try it!

OS9: list greetings <ENTER>
Good Morning
Welcome to OS-9 Dale!
OS9:

CMP

Every once in a while things go wrong while you are writing a file. Sometimes it's the computers fault. On other occasions, you may make a mistake. When you do, it sure is nice to be able to compare two files. A good compare utility can point you to a problem rather quickly. And, with the OS-9 CMP utility command you can compare any two OS-9 files.

CMP compares each byte in a file to the corresponding byte in another file. If it finds any differences, it gives you their location. It also displays the value of the byte stored at the disputed location in both files.

Let's deliberately create two files that are different:

OS9: build key <ENTER>
? This is a key. <ENTER>
? <ENTER>

OS9: build knot <ENTER>>
? This is a knot. <ENTER>

? <ENTER>

OS9: list key <ENTER>
This is a key.

OS9: list file2 <ENTER>
This is a knot.

Now let's see if the computer can uncover our plot.

OS9: cmp key knot <ENTER>

Differences

byte	#1	#2
0000000B	65	6E
0000000C	79	6F
0000000D	2E	74
0000000E	0D	2E

Bytes compared: 0000000F
Bytes different: 00000004

file2 is longer

COPY

Computers will play. Sometimes, they even crash. And when they do, it sure is nice to have a backup copy of your important files. OS-9 has a utility command to do this important job.

The OS-9 COPY utility reads the information stored in one file and writes an exact copy into another file.

When you run the COPY utility command, the first filename you type on your command line must exist. COPY creates a file with the second name you type. Then, it writes all of the information stored in the first file into the new file. COPY knows when it has read all of the information in the first file because it receives an end of file signal from OS-9.

Because COPY reads and writes large blocks of data, it does not do any line editing. You can tell COPY how many bytes you want it to read and write on each pass by using the the OS-9 memory modifier character, #, on your command line.

For example, if you want COPY to read 20,000 bytes from the first file and write them all to your new file during each pass, you could use the following command line.

OS9: copy #20K Spell Clone_of_Spell <ENTER>

You'll save a lot of wear and tear on your disk drives, not to mention a lot of time, if you request a lot of memory when you run the COPY utility. If you are doing a lot of single drive copies, you will thank your lucky stars that OS-9 lets you use the memory modifer.

Lets give it a try. Type:

OS9: copy KISS_12 KISS_12_BackUp #15K <ENTER>

This command line copies a file named KISS_12 in your current data directory to a file named KISS_12_BackUp in the same directory. When COPY finishes its work, you will see both files if you list your current data directory.

OS9: copy /d0/cmds/dir /d1/cmds/dir <ENTER>

This command line shows how you can COPY a file from one directory to another. It also shows you that COPY can recognize a complete OS-9 pathlist as well as a filename.

To tell COPY that you need to do a single drive copy, you must type the "-s" option in your command line. When you choose this option, COPY displays a message each time you need to change disks.

Here is an abbreviated session using COPY's single drive copy option.

OS9: copy chpt1 chpt1_backup -s #20K <ENTER>
Ready DESTINATION, hit C to continue: c
Ready SOURCE, hit C to continue: c
Ready DESTINATION, hit C to continue: c
Ready SOURCE, hit C to continue: c
Ready DESTINATION, hit C to continue: c

DEL

Many office workers have a serious problem. They are afraid to throw anything away. As a result they fill up drawer after drawer, file cabinet after file cabinet. Only when the room is full do they get around to purging their files.

If you are using an OS-9 based computer, you don't have any excuse for leaving extra files on your disks. Using the DEL utility command, you can easily delete any unwanted files.

There's only one catch. Before you can delete a file, you must have permission to write to that file. This means that you must either be the owner or the superuser.

The DEL utility lets you use one option, -x. If you type a -x in your command line, OS-9 will look for the file you named in your

current execution directory. Normally, it looks for the file in your current data directory.

There is one other thing you can't do with DEL. You cannot delete a directory file. To get rid of a directory you must use the OS-9 DELDIR utility command.

Here we go again! That statement is not 100 percent true. You can delete a directory using DEL if you have a lot of patience.

To do it, you must first delete all of the files in the directory. Then, you must run the ATTR utility command to change the 'd,' or directory attribute, to a "-d." When you set the "-d" attribute, you tell OS-9 that the file is not a directory. Since it is now only a file, you can go ahead and delete it. Fortunately, Microware wrote DELDIR to do all of this for us.

Here's a command line that uses the DEL utility:

OS9: del chapter_1 chapter_2 chapter_3 <ENTER>

As you can see, DEL lets you delete more than one file at a time. It can also recognize a complete OS-9 pathlist as well as filenames in your current data directory. Give it a try!

OS9: del /d1/chapter_11 <ENTER>

You're probably skeptical, so we'll attempt to prove that DEL does its job. First, list a directory. For example:

OS9: dir /d1 <ENTER>>

```
  Directory of /d1    15:46:57
Chapter_1     Chapter_2     Chapter_3
```

Now, run the DEL utility:

OS9: del Chapter_1 <ENTER>

Now, list the directory again:

OS9: dir /d1 <ENTER>>

```
  Directory of /d1    15:47:55
Chapter_2     Chapter_3
```

DUMP

If you want to get dizzy, try this. Use the LIST utility command on one of the binary object code files in your /D0/CMDS directory. For example:

OS9: list /d0/cmds/dir <ENTER>

Nine times out of 10 your screen will go bonkers. This happens because the LIST command displays not only ASCII characters, but also non-printing control characters. When your Color Computer screen or terminal receives these control codes, its hard to tell what will happen.

Yet, there are going to be times when you need to know exactly which characters are stored in a file. That's why Microware and Radio Shack gave us the DUMP utility command.

You can use DUMP to display information read from the standard input path in hexadecimal notation. Since all hexadecimal notation is formed from ASCII characters, your screen can display them.

When you run DUMP, you will see eight bytes of data listed side by side on each line of your Color Computer screen. On most of the larger OS-9 computers, DUMP displays 16 bytes on each line.

In addition to printing the hexadecimal value of each byte read from the standard input path, DUMP also displays the ASCII value on the same line. If a byte is non-printable — a control character for example — DUMP prints a period in the place where its ASCII value would appear.

DUMP also shows you how far a byte is stored from the beginning of a file. This means that if you DUMP an executable file from your /D0/CMDS directory, you will be able to tell the exact offset of each byte in the file after it is loaded into memory. Remember, all OS-9 modules are stored in a file in exactly the same form that they appear in memory.

Since DUMP reads characters from the standard input path you can even "dump" characters from the keyboard. Give it a try. Type:

OS9: dump <ENTER>
Now is the time. <ESCAPE> or <CLEAR><BREAK>

```
Addr 0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 2 4 6 8 A C E
---- ---- ---- ---- ---- ---- ---- ---- ---- ----------------
0000 4E6F 7720 6973 2074 6865 2074 696D 652E Now is the time.
```

Isn't that cute? Now let's try a real file.

OS9: dump /d0/cmds/load <ENTER>

```
Addr 0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 2 4 6 8 A C E
---- ---- ---- ---- ---- ---- ---- ---- ---- ----------------
0000 87CD 0024 000D 1181 0C00 1201 C24C 6F61 .M.$........BLoa
0010 E404 103F 0125 07A6 8481 0D26 F55F 103F d..?.%.&...&u.?
0020 0626 7381                               .&s.
```

Here's an interesting twist to the DUMP utility command. There are some unadvertised options buried in the DUMP utility supplied by Radio Shack. Try these command lines, too. You'll be pleasantly surprised.

```
OS9: chd /d0/cmds
OS9: dump dir >/p
OS9: dump -h list >/p
OS9: dump -l mdir >/p
OS9: dump -l -h dump >/p
```

The -h option lets you display a dump of a file without a header. This means its output would work well as input piped into an intelligent filter.

The -l option prints the dump 16 bytes across. The normal Color Computer format puts eight bytes across the 32 column screen.

Notice also that since the DUMP utility writes its output to the standard output path that you can redirect it to another file or device. In the command lines above, we send several DUMPs to the printer.

LIST

Files wouldn't be much good if you couldn't look at them. That's why Microware gave us the LIST utility command.

UNIX has its CAT utility, OS-9 has LIST. The two are very similar.

Since LIST sends its outputt o the standard output path, you can use it to read the English language text in any file you name on your command line. And because you can give it more than one filename in a command line, you can read several files at a time. LIST runs until it reaches the end of the last file you name. When it receives an end of file signal from this file, it returns you to OS-9.

Go ahead, try it! First, with a single file and a complete pathlist:

```
        OS9: list /d0/startup <ENTER>
echo
t
                          *************************
                          **   Dale L. Puckett   **
                          **      DaleSoft       **
                          *************************
-t
setime 84 ; * start clock
date,t    ; * print date and time
load load
load utils1 ; * load most-used utilities
```

```
load utils2
tsmon /esterm&
load spell lk
printerr
unlink load
link shell
```

Because the LIST utility sends its output to the standard output path, you can redirect it to any file or device. Here's the command line I used to get the results of the first command line above into a file so that I could print it here.

OS9: list /d0/startup >kl

After LIST created the file named "kl," I copied it into this document using Dynastar.

I could just as easily have sent the file to the printer by typing:

OS9: list /d0/startup >/p

In fact, I could emulate a typewriter by typing:

OS9: list /term >/p ; * an on-line typewriter

Finally, let's give LIST a number of files to work on. Try something like this.

OS9: list chpt1 chpt2 chpt3 <ENTER>

Of course, the files you name on your command line must really exist. Just for fun, type one that doesn't and see what happens.

Remember, you cannot effectively list a file that contains non-printable characters. If you LIST a file that contains control codes, look out. You'll see a lot of garbage on your screen and as often as not you'll crash your terminal.

MERGE

Quite often you need to combine a number of small files into one large file. You can almost do the job with the LIST utility command discussed above. But if you do, you'll run into problems.

When you use LIST, the automatic line editing feature found in all OS-9 device drivers can change your file. For example, in many cases these drivers will add a linefeed character following each carriage return.

If this line editing takes place while you are trying to combine two files containing binary object code, the line editing will change the program. The program won't run. Out of need, the MERGE

utility command was born.

You must use MERGE when you need to combine a number of small files. This utility command reads each file you name in your command line and writes the information from it to the standard output path.

This means that if you redirect the standard output path to another file you can combine several smaller files into one large file. And, since MERGE does not edit the information it sends to the standard output path, you can use it to copy files that hold program modules.

Here's how you can merge two files. Try it with your own files.

OS9: merge /d0/cmds/ds /d0/cmds/pinterp >New_Dynastar <ENTER>

This command line will merge a copy of Dynastar which is stored in a file named "ds" with the PASCAL p-code interpreter, pinterp. It saves them in a file named New_Dynastar, which would be stored in the current data directory.

You should be aware of one problem with the MERGE utility. When you use it to combine two or more files that contain executable object code modules it changes the attributes of the new file and makes it non-executable. This means you cannot load the file and run it.

To fix the problem you must use the ATTR utility command on the new file and set the 'e' or "pe" options.

RENAME

Each file in an OS-9 directory must have a unique name. You really can't argue with that statement, but what happens if you give one of your files the wrong name. Then later, you run a program that can only use that name.

Our worst case scenario is definitely exaggerated, but it could happen. Besides, we're all human. Aren't we? And we do make mistakes. Don't we? Besides, when we do make a mistake naming a file, the OS-9 RENAME utility command gives us a way to correct it.

You can use the RENAME utility command to change the name of any file. However, you cannot RENAME a file unless you have permission to write to it.

Before you get any wild ideas, it is not possible to RENAME a device or a directory. Don't even try.

Do try a command like this.

OS9: rename outstanding for_sure <ENTER>

When you run this command line, OS-9 will look for a file named outstanding in your current data directory. If it finds it, it will change the name of the file to for_sure.

You can also give RENAME a complete pathlist to the file you want to rename. Like this:

OS9: rename /d1/speech soapbox <ENTER>

Notice that you did not need to give the complete pathlist to the new name when you typed your command line.

Rename is similar to the UNIX MV utility command.

TEE

Every once in a while you are going to want to look at the dump of a binary object code file, print a copy on your printer, transmit a copy to a friend who has signed on to your computer through a telephone line and modem, and make an insurance copy in a disk file at the same time. Demanding, aren't you? Sound impossible? It's not with the TEE utility command.

You can use the TEE utility command to copy anything on the standard output path to any number of files or devices. TEE also automatically sends the information it receives from the standard input path to the standard output path.

Because the TEE utility is a filter, a command that receives all of its information from the standard input path and writes all of its output to the standard output path, it may be used in a pipeline to send a listing to your terminal, printer and a disk file — or any number of destinations — all at the same time.

Here's how you do it.

OS9: list /d0/startup ! tee /p /d1/BOOK/scratch <ENTER>

This command line uses the list command utility to read a file named startup on device /d0. The exclamation point pipes the output of the list command into the input of the TEE command utility.

TEE then writes a copy of the startup file to your printer and a file named "scratch" in a directory named "BOOK" on a disk mounted in device /D1. It also writes a copy to the standard output path which is most likely your Color Computer screen or a terminal.

TEE can also be used to get a message to people working at other terminals on your computer. For example:

OS9: echo It's time for the meeting ! tee /t1 /t2 /t3 <ENTER>

The OS-9 Tee differs from the UNIX Tee in that it does not use the append option. The UNIX Tee will create a file and write to it if it does not exist, write over it if it does exist, or append to it if you use the "-a" option. The OS-9 Tee will report an error if you attempt to Tee to a file that already exists.

SUMMARY

OS-9 lets you do a lot with files. In this chapter alone you have been introduced to a dozen different commands used to build, copy and delete them. In Chapter 12 we look at commands designed to work with OS-9 directories.

CHAPTER 12

commands that work with directories

Much of the power of the OS-9 operating system can be traced to the fact that it uses a hierarchical file system. In English, this means that you can organize your files according to context. You can store oranges with oranges and apples with apples.

It's quite a job to organize your disk files. Further, it takes a lot of planning. If you have a hard disk with a seemingly endless amount of storage you may be tempted to do it the easy way. Don't!

If you start throwing files onto a hard disk in a random fashion, you will have a mess on your hands within two or three months. It will take forever to straighten out the mess.

Here are some rules that should help you get your disks organized right from the start.

1. The "root" directory on a device should only contain other directories. All files should be stored in directories.

2. Directories of a like type should be grouped in a master directory. For example, a "USERS" directory should hold a group of directories, one for each user.

3. Master directories are like "root" directories. They should contain only directories. Files must go only in subdirectories.

4. Files of a like type should be grouped together in a directory.

5. When you find that a directory listing contains more files than your screen can display, break that directory up into several smaller directories.

You are going to need a pretty hefty set of tools to do all this work. For starters, OS-9 gives you eight utility commands that you can use to manipulate directories. In this chapter you'll be introduced to:

chd	dsave
chx	makdir
deldir	pwd
dir	pxd

CHD

There's no question about the usefulness of the current data directory concept. Which one of these command lines would you rather type?

OS9: /d0/CMDS/list /D1/DALESOFT/ACCOUNTS/PAYABLE/January
OS9: list January

The first command line is very readable and very organized. Unfortunately, it is also prone to typos. The second is a piece of cake to use. Through the magic of the OS-9 CHD utility command, you can use the second form most of the time.

Microware wrote the CHD command to give you a way to change your working data directory. Since it is part of the OS-9 SHELL, you do not need to keep a copy stored in your /D0/CMDS directory.

Most OS-9 utility commands and application programs automatically look in your current data directory when they need to find text files, programs and other data. They also usually store data that they create in your current data directory.

CHX is to your working execution directory as CHD is to your working data directories. In other words, it gives you a way to change your working execution directory. It also is built into the OS-9 Shell.

When OS-9 is looking for a command that you have typed in a Shell command line, it looks first at its module directory in memory. If it finds a module with the right name, it links to it and executes your command immediately.

But, what happens when there is not a module in memory that has the correct name? You guessed it! OS-9 looks in your current execution directory. OS-9 knows that all files it finds there will contain code that it can load and run. It is up to you to make sure that directory lives up to its expectations.

Drive CHD and CHX for yourself! Here's the format.

chd /d1/SAMPLE_PROGRAMS <ENTER>

This command line makes a directory named SAMPLE_PROGRAMS which is stored in the root directory of a disk mounted in device /D1 the current data directory.

Let's try another one.

chd .. <ENTER>

If you run this command line immediately after you run the first, the root directory of the disk mounted in the drive named /D1 will become your current data directory. Why? Because "/D1" is the parent directory of the directory SAMPLE_PROGRAMS. And, ".." means "the parent of the present directory."

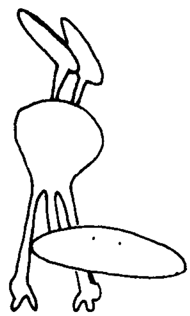
Let's do two more.

chx ../CMDS <ENTER>
chx /d1/BASIC <ENTER>

Get the idea? When you use the CHD and CHX utility commands with OS-9's anonymous directories, ' . ' and ' .. ' you will find it is very easy to crawl up and down your directory trees.

Here's one more sequence for the road.

OS9: chd /d1/TOUR_GUIDE <ENTER>
OS9: dir <ENTER>



```
   Directory of . 14:57:55

dict     Chpt1     Chpt2     Chpt3
Chpt4    Chpt5     Chpt6     Print_Book
```

DIR

Let your fingers do the walking! Without the yellow pages we would all be lost. The same goes for the OS-9 DIR utility command. If we didn't have this simple program, we wouldn't be able to find the data files we stored in our directories last week.

And just think, if we couldn't list the contents of our current execution directory, we would never be able to decide which program to run.

You can use the DIR utility command to display a listing of the names of all files that are stored in any of your directories. DIR sends these names to the standard output path which means that they will appear on your terminal unless you redirect them to another device.

DIR does have two options. If you type an 'x' on the command line, DIR will display a listing of your current execution directory.

You may also type an 'e' on the command line. If you do this, you will receive a complete description of all files in your directory. This extended directory also tells you the size of a file and its address on your disk. And if you look closely, you can tell who owns each file and what type of security has been set up for the file. If you really need to keep tabs on your employees, you can find out the date and time when each file was last modified.

How about a test drive? First, here's the command in its simplest form.

OS9: dir <ENTER>

```
     directory of . 20:34:08
print_Book   chpt6      chpt8      chpt4      Chpt18
Chpt19       adtoch17   adtoch19
```

This command line lists the name of each file stored in your current data directory. Now let's look at the current execution directory.

OS9: dir x <ENTER>

```
     directory of . 20:31:28
backup       binex      cmp        dcheck
deldir       sort       dsave      exbin
```

Now let's take a look at an extended listing of the same directory.

OS9: dir x e <ENTER>

```
     directory of . 20:25:20
Owner  Last modified  attributes sector  bytecount  name----
  0    83/09/01 1106  —e-rewr     56       4C2      backup
  0    83/09/01 1106  —e-rewr     5C       278      binex
  0    83/09/01 1106  —e-rewr     60      1019      cmp
  0    83/09/01 1106  —e-rewr     72      27C6      dcheck
```

etc . . .

In addition to the name, this format tells you who owns each file, when it was last modified and which attributes it carries. It also tells you how many bytes are in each file and the sector where each file is stored.

Our last two examples show you that you can also ask OS-9 for a directory listing by giving it a complete pathlist.

OS9: dir /d1/BASIC_PROGRAMS <ENTER>

OS9: dir /d1/BASIC_LISTINGS e <ENTER>

DSAVE

If you had to type in the many command lines needed to copy each file in one directory to another directory every day, you would soon look for a way to retire early. The job is especially complicated by the fact that you can only have one current data directory. This means you must type the complete path list for one of the command line parameters. If there are more than a half-dozen files in a directory, the job could take forever.

Never fear, OS-9 has a solution for you. The answer is DSAVE, a utility command that creates a procedure file that automatically copies all of the files in one directory into another directory.

When you run DSAVE, it generates command lines that copy files from your current data directory to a directory you name. It sends these command lines to the standard output path. This means that you can send its output to a procedure file in your current data directory, or pipe it into a Shell and copy the directory on-line.

If you save DSAVE's output in a procedure file, you can edit the file. This could come in handy when you only want to save part of a directory. On the other hand, it is handy to be able to save a directory on-line using the second method. And, it's fun to watch. Name your poison.

Here's another nice thing about DSAVE. If it finds a directory name, it automatically generates a MAKDIR command and changes the working data directory for you. Because it has this feature, DSAVE can also be used to copy all the files stored on a disk mounted in one device to a disk mounted in another device.

You can use six options in a DSAVE command line. They are: -b, -b=<filename>, -i, -L, -m, and -s.

"-b" lets you make a new system disk. To do this, it copies your OS9Boot file from the original disk to the new disk. It also links to it.

"-b=<path>" also lets you create a new system disk. But, it reads your new OS9Boot file from the file you name rather than from the OS9Boot file on your old disk.

"-i" gives you a way to indent DSAVE's output at each directory level. You can also use it to display a hierarchical listing of all directories on a disk.

"-L" gives you a way to only copy the files stored at your current directory level.

"-m" keeps DSAVE from automatically including MAKDIR commands in your procedure file. You will need to use this option when you try to copy a set of directories from a disk to another disk that already has those directories.

And finally, "-s<integer>" gives you a way to tell OS-9 how much memory you want it to use when it copies your files. Note here that the integer value you type next to the 's' should be the number of kilobytes of memory you need rather than the number of pages.

Let's take a test drive through a typical DSAVE. First we'll need to tell OS-9 which directory we want to copy. We do this by naming it as the current data directory. Remember, we do this with the CHD utility command.

OS9: chd /D0/SYS <ENTER>

Then, we must run DSAVE to create a procedure file.

OS9: dsave -s20 /D0 >CopyIt <ENTER>

Now, we need to give OS-9 the name of a directory it can use to store our files. We use the CHD utility command to do this also.

OS9: chd /D1/SYS <ENTER>

Finally, we can run the procedure file that DSAVE created for us. Remember, we stored it in a file called CopyIt. That file is stored in the directory named /D0/SYS.

OS9: /D0/SYS/CopyIt <ENTER>

Here's a listing of the procedure file CopyIt which DSAVE created for us.

```
t
tmode .1 -pause
load copy
Copy #20K /d0/SYS/password password
Copy #20K /d0/SYS/motd motd
Copy #20K /d0/SYS/mkcpy mkcpy
unlink copy
tmode .1 pause
```

As you can see, if you are trying to copy a directory with 20 or 30 files — or an entire disk — DSAVE can save you a lot of work.

You can also use DSAVE interactively if you pipe its output directly into a Shell. Here are two alternate forms you can use. Give them a try when you have some time.

OS9: dsave /d1 /d0 ! Shell <ENTER>

This command line will copy all of the files stored on a disk mounted in device /d1 to a disk mounted in device /d0. It will automatically create any directories it needs.

OS9: chd /d1/SPELL
OS9: dsave /d1 ! (-x chd /d1/NEW_SPELL)

This sequence will let you copy all of the files in a directory named /d1/SPELL to another directory named NEW_SPELL on the same device.

MAKDIR

When you get ready to organize your disks, you'll find that you need a way to create new directories. OS-9 gives you a utility command to do the job.

To create a new directory on an OS-9 disk, you need only run the MAKDIR utility command. Running MAKDIR is as simple as giving OS-9 the name of the directory you want it to make.

MAKDIR does suffer from one "gotcha." You cannot create a directory in a directory or on a device unless you have permission to write in the parent directory. Do you see now why it took us so long to cover the ATTR utility command?

Here's another tip. It's a good idea to capitalize all the letters in a directory name. This makes it stand out from other filenames.

It's time to go to work. Try this:

OS9: makdir /d1/TOUR_GUIDE/ <ENTER>

This command line creates a directory named TOUR_GUIDE in the root directory of device /D1. That was fun! Let's make another.

OS9: makdir HOMEWORK <ENTER>

This command creates a directory named HOMEWORK in your current data directory. What's that? You don't know which directory you're in. Stay tuned for PWD. You'll soon learn that it's a good idea to find out where you are before you start making new directories.

Here's a MAKDIR command line that uses an anonymous parent directory.

OS9: makdir ../MUSIC_LIBRARY <ENTER>

This command line uses OS-9's anonymous directories to create the directory MUSIC_LIBRARY in the parent directory of the current data directory. I'll bet you didn't realize that you would

be studying genealogy when you bought OS-9.

Here's some trivia that just may come in handy when you graduate into the OS-9 hacker ranks. When OS-9 creates a new directory file it contains only the names of the two anonymous directories, '.' and "..".

Here's another important point. When MAKDIR creates a directory it turns on all of the directory file's attributes. This means everyone running your computer has permission to use the new directory.

MAKDIR is very similar to the UNIX Mkdir utility command.

PWD

It's very easy to get lost when you first start to work on a computer that uses hierarchial directories. Because of this, OS-9 gives you two utility commands that can show you your location in the directory tree.

Let's review for a moment. Remember, the current data directory is where you are currently storing your data files. You can read any file in this directory by giving OS-9 its filename. However, when you want to read a file stored in any other directory, you need to give OS-9 a complete pathlist.

The current execution directory is similar. You should always store the files that hold the object code of the programs that run on your computer in this directory.

The PWD utility command gives you a way to display the complete pathlist to your current data directory. What does all this mean? Stated simply, it means that OS-9 can tell you where you are when you get lost in its file system.

The pathlist displayed by the PWD utility will take you all the way back to the "root" directory of the disk mounted in the device — disk drive — that holds your current data directory.

PXD does essentially the same thing for your current execution directory. It displays the complete pathlist for your current execution directory — all the way back to the root directory of the disk mounted in the device that holds your current execution directory.

A few examples should make things clearer.

First, let's make sure we know where we are.

OS9: chd /D1/TOUR_GUIDE/CAPTIONS

Now, let's ask OS-9 where we are.

OS9: pwd
/D1/TOUR_GUIDE/CAPTIONS

How about that! Now let's play around some more with those tricky anonymous directories.

OS9: chd ..

That command line should have put us in a directory named TOUR_GUIDE stored in the root directory of the disk mounted in drive /d1. Let's check it out.

OS9: pwd
/D1/TOUR_GUIDE

OS-9 two! Humans two! We're on a roll. Let's see if PXD can do its job.

OS9: pxd
/D0/CMDS

Mission accomplished!

SUMMARY

In this chapter you have been introduced to the tools you'll need to effectively manage your OS-9 disk files. Have some fun now! Create a few directories on a disk. Then, crawl up and down your new directory tree for a while.

Join us in Chapter 13 when you get tired. We'll be taking a look at the utility commands you'll need when you get ready to create a new system disk or copy an old one.

CHAPTER 13

commands used to create or copy os-9 disks

After you learn how to manage OS-9 files and make your Color Computer dance to your drum, you'll still want more. That's human nature.

If you're a computer hacker, don't worry. You're just like everyone else in the world. All you want to do is customize your computer so that it will have your own personal signature.

After you learn the commands introduced in this chapter, you should be able to customize forever. In Chapter 16 we pursue this avenue in great detail. For now, we hope you will enjoy your tour of:

backup	load
cobbler	os9gen
dcheck	save
format	verify

BACKUP

In Chapter 11 we showed you how you can make a backup copy of a disk file using the OS-9 COPY utility command. COPY is a very useful tool, but it would take you an eternity to individually copy each and every file on a 5-inch disk drive — let alone an 8-inch drive or hard disk system.

We partially solved the problem in Chapter 12 when we showed you how to use the OS-9 DSAVE utility command to automatically COPY all the files in a directory or on a disk. You'll need to use DSAVE when you want to copy a directory from one

type of disk to another — say from single-sided to double-sided, or from 5-inch to 8-inch, for example.

However, when you only need to copy the files on one disk to another disk of the same size and type, OS-9 gives you a handy utility command that lets you do the job fast.

You should use the BACKUP utility command whenever you need to copy all of the information on one disk to another disk of the same format.

BACKUP copies each sector on one disk onto the same sector on the other disk without stopping to see what was in the sector. It pays absolutely no attention to the file structure of your disk. It doesn't care what is stored on your disk. That's why it is so fast.

Because BACKUP doesn't check, it is up to you to make sure that your source and destination disks are the same size and that they have the same format. This means they must both have the same number of tracks and the same number of sides. Also, if the source disk has been formatted to be single density, the destination disk must also be single density.

When you run BACKUP, you usually give it two device names. It will then copy everything from the first device to the second. However, you can use a shorthand form of the command and skip typing the device names. When you do this, BACKUP asks you if you want to BACKUP device /D0 to device /D1. If you neglect to type the second device name, BACKUP asks you if you want to do a single disk BACKUP.

When you are doing a single disk BACKUP, OS-9 will tell you when to change your disks. It works just like the single disk copy we described in Chapter 11.

You can use four options on your command line when you run BACKUP. They are: e, s, -c and #nK.

'e' gives you a way to stop the BACKUP if there is a read error

's' instructs OS-9 to tell you when to change your disks while you are doing a single drive BACKUP.

"-v" tells OS-9 not to verify the new disk after it copies all the sectors from the original disk.

"#nK" tells OS-9 to use 'n' thousand bytes of memory each time it reads a block of data from the first disk and writes it to the second. In general, the more memory you give BACKUP, the faster it will run.

Let's take a spin.

backup /d1 /d3 <ENTER>

This command line lets you make an exact copy of the contents of the disk mounted in device /D1 on a disk mounted in device /D3.

Let's try a single disk BACKUP next.

backup /d0 #20K <ENTER>

Ready to BACKUP from /D0 to /D0 ?: Y
Ready DESTINATION, hit a key:
DISKNAME
is being scratched
OK ?: Y
Ready SOURCE, hit a key:
Ready DESTINATION, hit a key:
Ready SOURCE, hit a key:

etc ... until finally

Ready DESTINATION, hit a key:
Number of sectors copied: $0276
Verify pass
Number of sectors verified: $0276

The command line above let you make a single drive backup of the disk mounted in device /D0. OS-9 told you when to change your disks. Changing disks like that gets old fast. But, it works when you only have one drive or your second drive is out being repaired.

Notice also that OS-9 used 20K of memory each time it copied part of the disk. It did this because you used the OS-9 memory modifier in your command line.

Here's another form of the BACKUP command.

OS9: backup -v <ENTER>

This command line makes an exact copy of the disk mounted in device /D0 on a disk mounted in device /D1. However when you use the "-v" option, OS-9 does not verify the data that is written on the new disk.

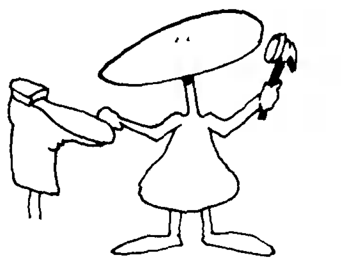
The "-v" option will speed up your BACKUPs, however you will have more peace of mind if you let OS-9 verify your data as it is copied. The choice is yours.

Here's one more for the road:

OS9: backup <ENTER>
Ready to BACKUP from /d0 to /d1 ?: y

MYDISK is being scratched
OK ?: y
Number of sectors copied: $04FA
Verify pass
Number of sectors verified: $04FA
OS9:

COBBLER



If you are going to use a disk to start up your system, it must contain a special file named OS9Boot. If you want the OS9Boot file on your new "boot" disk to be the same as the one you have been using, you can use a shorthand command supplied with the OS-9 utility command set. This command is called COBBLER.

What do cobblers do? Or, should I ask what did cobblers do? You got it, they make new "boots." The authors of UNIX and OS-9 were all punsters.

Getting serious again, the OS9Boot file cannot be segmented. This means that when you create this disk, there must be a block of free space on the disk big enough to hold your entire OS9Boot file. For this reason, most people like to run COBBLER on a freshly FORMATTED disk.

When you run the COBBLER utility command, it writes a new OS9Boot file on the disk you name in your command line. This new file will contain all of the modules that were loaded into memory the last time you started OS-9.

COBBLER has a serious "gotcha" too. You can only use it on an OS-9 Level I system. If you try to run COBBLER on a Level II system, it will not work.

If you own an OS-9 Level II based computer, don't worry. You can use the OS9Gen command to do the same job.

Let's give it a try.

OS9: cobbler /D1 <ENTER>

This command line will write a copy of your last OS9Boot file on the disk you have mounted in device /D1. It's as simple as that.

Here are a few notes about the Color Computer version of COBBLER. It writes the OS-9 Kernal on the first 15 sectors of track 34. After it does this, it removes these sectors from the disk's allocation map. If a file has already been stored on this track, COBBLER will send you an error message. The DOS command in the Disk Extended BASIC ROM loads track 34 into memory when you start OS-9.

Sometimes, it is possible to confuse OS-9. For example, you could accidentally remove a disk from a drive while a file is still open.

Then later, after you have put another disk in the same drive, you close the file. Unfortunately, OS-9 doesn't have any way to know that you switched disks. It will write the file in the same location on the new disk, thinking that it is still writing on the first disk.

If you're confused, how do you think OS-9 will feel when it tries to read the disk structure on that second disk.

Besides, if you're a nervous Norvus, you probably always need to be told that the data you have stored on your disks is OK. OS-9 has a utility command that can do this for you.

Sometimes you will find that you cannot read a file. Yet, everything appears to be OK. You've checked to see if the file is a directory file; it isn't. You've checked to see if it is an executable file; it isn't. What next?

One thing you can do is check the file structure of your disk. If your only experience up to now has been with Color Computer Disk Extended BASIC, you're in for a treat. OS-9 gives you a command that you can only wish you had with your Disk Extended BASIC! Enter DCHECK.

DCHECK lets you verify the file structure of any disk mounted in any drive on your system. Don't be confused by the $0005A0 sector count. That's 1440 in decimal. I use 40 Track, double-sided, double-density drives.

If you are using a system with two drives, try entering:

DCHECK /d1

You should see a listing similar to this:

Volume - 'Rainbow_Articles' on device /d1
$00B4 bytes in allocation map
1 sector per cluster
$0005A0 total sectors on media
Sector $000002 is start of root directory FD
$000A sectors used for id, allocation map and root directory
Building allocation map work file...
Checking allocation map file...

'Rainbow_Articles' file structure is intact
4 directories
7 files

If you own the disk in device /D1 you can breathe a sigh of relief. Let's consider some other uses for DSAVE.

Perhaps you've just developed a serious software package designed to count the number of hairs on a balding head. The disk that contains your program and it's necessary modules and data files includes a number of files that are stored in several subdirectories. Since your program is dedicated to counting, you want to tell potential customers exactly how many files and directories are on the disk.

You need a quick count of every file on the entire disk, but you are afraid it would take days to trace all the directories. No problem; its time to let DCHECK go to work. DCHECK has an option that can give you the exact information you need.

If your disk has as many subdirectories as the Tandy OS-9 System Disk, you will find an unknown number of files stored in an unknown number of directories. Use the DCHECK command with the -s option and you should see something like this:

DCHECK -s /d1

4 directories
59 files

Other DCHECK options that you may want to experiment with include: -b, -p, -w=<filename or pathlist>, -m and -o.

"-b" lets you check the structure of a disk without listing the unused clusters.

"-p" prints the complete pathlist to each cluster that may have a problem.

"-w=<filename>" tells OS-9 to store DSAVE's work files in a file named filename. You can also give DSAVE a complete OS-9 pathlist with the -w option. For example, -w=/d0/SYS/scratch.

"-m" tells OS-9 to keep the allocation map work files that DSAVE writes.

"-o" tells OS-9 to print all valid DCHECK options. For all practical purposes it is a HELP command for DCHECK.

You should note that DCHECK cannot process a diskette with directories more than 39 levels deep. Who cares? How many times are you going to wind up at level 39 with single-sided, 35 track drives. If you ever find yourself that far out on a limb, you've probably overorganized your directories.

Before you can store any programs or data on an OS-9 disk you must initialize it. When OS-9 initializes a disk, it writes a special sector format to every sector on the disk. Then, it verifies each sector to make sure that the disk surface is OK. After it has done all this, it writes an allocation map, root directory and identification sector on track zero.

This process sounds complicated. But, breathe a sigh of relief, it is totally transparent to you because OS-9 gives you a handy utility command to do the job. All you need to remember is the fact that you must FORMAT a disk before you try to use it in your OS-9 based computer.

When you run FORMAT, OS-9 can tell what type of disk drives you own by reading a table of values in your device descriptor. When you are in the mood to do something different however, you can also tell OS-9 how to FORMAT your disk by giving it information in your command line.

The FORMAT utility command can recognize seven command line options. They are: s, d, 1, 2, 'number', :number:, and "name".

If you type an 's' in your command line you are telling OS-9 that you want the disk being formatted to be single density. If you type a 'd,' FORMAT will know that you want your new disk to be a double-density disk.

When you type a '1' FORMAT takes it to mean that you want to format only one side of the disk. Likewise if you type a '2,' FORMAT will initialize both sides of a disk mounted in a double-sided drive.

You can also tell FORMAT how many tracks you want on the disk, the sector interleave value and the name of your new disk. To do this, you type the decimal number of tracks you want between single quotes, the sector interleave value you want between colons and the name of the disk between double quotes.

Here's how you do it.

OS9: format /d0 2 D "RAINBOW" '77' <ENTER>

This command line tells FORMAT to initialize the disk mounted in device /D0. Further, it tells FORMAT that you want it to initialize both sides of the disk and that you want the new disk to have a double-density format.

You have also told FORMAT in your command line that you want to name your new disk "RAINBOW" and that you want to initialize 77 tracks. This means that device /D0 must be an eight-inch drive.

Play it again, Sam!

OS9: format /d1 S 1 <ENTER>

This command line tells FORMAT that you want to end up with a single-sided disk that has a single-density format.

Let's run FORMAT once more. This time we won't tell it what we want in our command line. We'll let it read our device descriptor. You will be able to see the report FORMAT gives before it goes to work.

OS9: format /d0 <ENTER>

```
                 FORMAT 1.1

          TABLE OF FORMAT VARIABLES

Recording Format:                    MFM
Track density in TPI:                 48
Number of Cylinders:                  77
Sector Interleave Offset:              3

Disk type:                             8
Sectors/Track on TRK 0, Side 0:       16
Sectors/Track:                        28

Formatting on drive /d0
Y (yes), n (no), or q (quit) y
Ready: y
Disk Name: RAINBOW BOOK
```

You need to type 'y' — for yes — twice. Then, FORMAT will go to work. It will initialize the disk mounted in drive /d0 and then stop and ask you for the name you want to give your new disk.

After you type the name of your new disk, you will see FORMAT display each track number. It prints a new number each time it verifies a track.

Finally, FORMAT tells you how many sectors it has formatted by displaying a good sector count.

GOOD SECTOR COUNT = $860

If you are using OS-9 on the Color Computer, you will find that FORMAT will only initialize single-sided, double-density disks. You'll also soon learn that this disk will have a non-standard format that holds 18 sectors on each track. This means you can store more information on a Color Computer OS-9 disk. A standard double-density OS-9 disk only stores 16 sectors in each track.

The Color Computer FORMAT Command does let you format

a disk with 40 tracks if you own 40 track drives and have modified the device descriptor that points to the drive you are using. We show you how to do this in Chapter 16.

LOAD

When you type the name of an OS-9 program in a Shell command line, OS-9 looks for a module with that name in its module directory. If it finds it, it links to the module and runs the program.

If OS-9 cannot find the name of your program in its module directory, it looks for a file with the same name in your current execution directory. If it finds it there, it loads that file into memory, links to the module and runs your program.

This means that if a module is already in memory, OS-9 can run your program sooner. You won't need to wait for it to load from a disk. To get a module that is stored in a disk file into memory you must use the OS-9 LOAD utility command.

When you run the LOAD utility command, it reads each and every module that is stored in the file you named in your command line and writes it into memory. After it has done this, it places the name of each of those modules in the OS-9 module directory.

I can hear your question now. What happens if I try to LOAD a program that is already in memory.

If you try to do this, LOAD will use the module that has the highest revision number. This means that if you have an old version of a program in your OS9Boot file or in read only memory, you can replace it by LOADing another module with the same name, but a higher revision number.

Here's the syntax for a Shell command line that uses LOAD.

OS9: load DynaSpell <ENTER>

You must make several tradeoffs when you decide which modules you want to LOAD in memory.

The alternative to using LOAD is to not load your utility command modules. Rather, you can just call each utility from the Shell command line and let OS-9 load them each time.

Let's looks at the two different approaches. First:

OS9: load list
OS9: list file.one
OS9: list file.two
OS9: list file.three
OS9: unlink list

And second:

OS9: list file.one
OS9: list file.two
OS9: list file.three

What are the tradeoffs? If you use the first approach you must remember to use the UNLINK utility command after you are finished with the LIST utility. Once you LOAD a module, it will remain in memory until you UNLINK it. If you forget to UNLINK a module, it may cause memory fragmentation.

If you use the second list of commands, OS-9 will need to go out to your current execution directory and LOAD the file named "LIST" each time you ask it to LIST a file. You will hear the drives LOAD "LIST" three times.

With a short utility file like "LIST" it probably doesn't make much difference. But what if the file is long, like ASM or BASIC09? You'll have to make your own decision.

OS9GEN

Each disk that you plan to use when you start up your OS-9 computer must contain a file named OS9Boot. On a Level I system you can use the COBBLER utility command to make a new "boot" disk.

But, here's the catch. You cannot use the OS-9 COBBLER command to make a new system disk if you change your OS9Boot file. To change your OS9Boot file, you must use a special OS-9 utility command named OS9Gen. You must also always use OS9Gen instead of COBBLER on all OS-9 Level II computers.

The OS9Gen utility command creates a new "OS9Boot" file each time you run it — even though you can use it to make an exact copy of your present boot file.

OS9Gen's main use however, is to create a new OS9Boot file that contains new or additional modules. It can also be used to make an OS9Boot file that has had modules taken out of it.

When you run OS9Gen, you must give it the the name of the device that holds the disk you want to install the new OS9Boot file on. You do this by including the device name in your command line.

You must also tell OS9Gen where it can find the files that hold the modules you hope to install in your new OS9Boot file. Oh well, they say a picture is worth a thousand words. Let's test the theory.

OS9: os9gen /d1 <ENTER>

/d0/os9boot <ENTER>
<ESCAPE> or <CLEAR><BREAK>

These lines tell OS9Gen to install the file OS9Boot that is stored on the disk mounted in device /D0 on the disk mounted in device /D1. It effectively does the same thing as the COBBLER utility command. However, it gets the modules from the file /D0/OS9Boot rather than from memory.

Here's another OS9Gen command sequence.

OS9: os9gen /d1 <ENTER>
/d0/os9boot <ENTER>
/d1/new_video_drivers <ENTER>
/d1/new_modem_drivers <ENTER>
<ESCAPE>

This sequence of commands makes a new disk with an OS9Boot file that contains all of the modules stored in the file OS9Boot on device /D0, plus two new modules. It gets the new modules from files named new_video_drivers and new_modem_drivers on device /D1.

Since OS9Gen reads its input from the standard input path, you can redirect its input to a file that contains a list of names. This will help eliminate problems from typos. In fact, you'll want to run OS9Gen this way when you need to put a long list of new files in your new OS9Boot file. Several sample procedures in Chapter 16 use this approach.

SAVE

Every once in a while you will need to use the OS-9 Debug command to change a module while it is in memory. After you do this a few times, you'll probably want to keep a copy of the new version in a disk file. It beats typing Debug commands. The OS-9 SAVE utility command is the tool you need to do the job.

The SAVE utility command gives you a way to save a copy of the data in a memory module in a disk file. Before you run SAVE you must make sure that the name of the module you are trying to save is in the OS-9 module directory.

If you give SAVE a filename it will store your module in a file with that name in your current data directory. If you want, you may also give SAVE a complete OS-9 pathlist. SAVE always stores the module in a file with the first name that follows it in your command line.

You may save one or more modules in each file. You ask the SAVE utility to put more than one module in a file by giving it a list of names in your command line.

Go ahead, give it a try.

OS9: save /d0/CMDS/NewStrip NewSTrip <ENTER>

This command line SAVEs a copy of a module named New-Strip in a file named NewStrip in the /D0/CMDS directory. Note that the first name after the word SAVE in the command line is always used as the pathlist to the file you are saving.

Here's another SAVE command line.

OS9: save /d0/CMDS/directions North East South West <ENTER>

This time our utility should SAVE four modules — North, East, South and West. It should store them in a file named directions in the /D0/CMDS directory. As we said earlier, SAVE can store one module, or any number of modules in one file.

VERIFY

Some utility commands act like good insurance policies. The OS-9 VERIFY utility falls into this category. It lets you know that the data within a module stored in a disk file is still valid.

But, VERIFY also has another function. It gives you a way to correct the CRC of a module that you have modified.

When you modify a module in memory and then SAVE it, the file you create will contain a module with a bad CRC. This happens because the CRC bytes that were generated for the module in its original form will not agree with the CRC of the modified module.

Yet, the original CRC bytes will be written to the disk file when you save the modified module. The consequence comes later when you try to LOAD the file containing the modified module. Always remember this. When a module has a bad CRC, OS-9 will refuse to load it into memory.

You can use the OS-9 VERIFY utility command to insure that the module header and CRC of all modules in a file are alright. VERIFY reads data from the standard input path and sends its output to the standard output path. This means that you must usually redirect its input so that it comes from the file you need to VERIFY. If there are any problems, VERIFY tells you by writing a message on the standard error path.

You can use the update 'u' option when you want VERIFY to send a new module with a corrected CRC to the standard output path. However, VERIFY will only update a file like this if you ask it to. When you do update a CRC with VERIFY you will need to redirect the output of the VERIFY utility as well as the input.

Let's give it a try.

OS9: verify <Speller <ENTER>
Module's header parity is correct.
Calculated CRC matches module's.

This command has VERIFY check a file named Speller, which is stored in your current data directory. Notice that you must redirect the input of VERIFY to the file containing the module you need to VERIFY.

Here's how you update a modules CRC.

OS9: verify <Bad_file >Good_file u <ENTER>

This time we asked VERIFY to update the CRC of the modules in a file named Bad_file. It saves the modules with a corrected CRC in a file called Good_file. It is the 'u' option in the command line that tells VERIFY to update the CRC bytes. Notice that we can redirect both the input and output with VERIFY since it reads from the standard input path and writes to the standard output path.

When we typed this command line, we assumed that both files were stored in the current data directory. Did you notice that we did not type a complete pathlist. Here's a reminder — a complete pathlist always begins with a slash.

Here's another trick. Study this command line:

OS9: merge /d0/cmds/dir ! verify

Try it and see what happens. Do you know why it works?

SUMMARY

In this chapter you have been introduced to a set of commands that you will need when you start creating new system disks. If we have awakened your appetite, stand by. In Chapter 16 we show you a number of procedure files where you can use these utility commands.

Practice awhile. Then, join us in Chapter 14 for a look at a few utility commands that will let you change a few of parts of OS-9 in memory.

CHAPTER 14

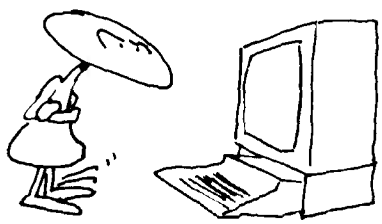
commands that change the system

Often, when you complain to other people that you "don't like this," they just look at you and tell you to do something about it. They make a good point.

How many times have you complained about your computer because it's always doing something you don't like. Maybe it beeps at you every time you make a mistake. Or, maybe it scrolls your carefully written prose off the screen while you're still trying to review it.

Those are just two of the problems you could fix if you knew how to change your operating system. Here's a philosophy to pursue. If you can't beat the system — at least use it to your advantage.

This chapter is about using your operating system to your advantage. Before we're through you'll meet eight more OS-9 utilities — commands that let you change the way your system responds to your beck and call. You'll learn about:



link	tmode
login	tsmon
setime	unlink
sleep	xmode

LINK

Do you remember when we talked about OS-9 memory management back in Chapter 3. You had a skeptical look in your eye when we told you that OS-9 always knows when you're finished with a module?

The magic comes from a single byte in the OS-9 module directory that holds a link count. That byte tells OS-9 how many programs are using a module at any given time. It works like this.

Before a process — remember, we call a program that is running a process — can use a module, it must link to it. When this link is made, OS-9 increases the value of the link count by one. When a program no longer needs the code in a module, OS-9 decreases the link count by one.

If you keep decreasing a counter, its value will eventually become zero. When a link count reaches zero, OS-9 knows that your process is finished with the module and releases it. OS-9 also removes the module's name from its module directory and releases any memory the module had been using. As soon as this memory has been released, another program can use it.

If you type the name of a command that is not in the module directory, OS-9 loads it and links to it — one time. The link count is set to one at this time.

When your program finishes its work, it releases the module and the link count becomes zero. OS-9 takes it from there and the module disappears from the module directory.

Now lets look at an example where you have LOADed a module into memory in anticipation of using it. We'll assume that no one else is using this module, so after you LOAD it the link count should be set to one. OS-9 always LINKs to a module when you LOAD it. Effectively, it has incremented the link count to one so that the module can stay in memory until it is needed.

What do you think happens when you run a program that uses this module? Follow this closely. First, the program that is using the module links to it. This means the link count is increased by one. It should now have a value of two.

If everything proceeds properly your program will eventually finish with the module. When it releases the module, the link count will be decreased by one. It should be equal to one. Since it has not reached zero, OS-9 leaves it in memory.

This all means that if a program is using a module and it knows that it is going to need it again soon, it can LINK to it. This insures that the link count never reaches zero and the module will stay in memory until its needed the next time.

To recap, LINK locks a module into memory. It also increases the link count of the module by one. When you use LINK on an OS-9 Level II system, it moves (actually, the proper terminology is "maps") the module you are LINKing into the memory area of the process that called it.

Here's how you would LINK to a module named "ls."

OS9: link ls <ENTER>

There are several things you should know when you use the OS-9 LINK utility command. First, you must always be sure that you UNLINK a program module after you use it. If you don't, you run the risk of fragmenting your memory. To see why, try the following command sequence:

OS9: load asm
OS9: load dir
OS9: dir
OS9: asm testfile
OS9: unlink asm
OS9: mdir e

Look closely at the memory map printed by the module directory utility. You'll notice that you have two rather large chunks of memory free with a small module containing a few hundred bytes sitting in the middle. You may have 120 pages of memory free, but it is split into two pieces only 50 pages long.

The BASIC09 module contains about 23,000 bytes of 6809 object code — that's more than 90 pages of memory. Since OS-9 Level I requires that all program modules be loaded into contiguous memory, this means you couldn't load BASIC09 into memory if your memory was fragmented like this. To get rid of this fragmentation you must UNLINK the DIR utility.

What's the answer? Just be careful and make sure you UNLINK a module when you are finished with it. For example:

OS9: load list
OS9: list file.one
OS9: list file.two
OS9: list file.three
OS9: unlink list

LOGIN

We'll start this section by assuming that you are a bona-fide user on your office computer. You even have a new password from the boss.

Your LOGIN name is the name OS-9 uses to identify you when you sign on. A password ensures that no one else can use your name to tamper with your files.

Review the short tutorial near the end of Chapter 4 to learn more about this sign on procedure. OS-9 uses a utility command named LOGIN when you sign on.

If you are responsible for the security of data stored on the computer system in your office or shop, you will love LOGIN. This OS-9 utility command gives you a way to ensure that employees can only work with their own files.

You can run LOGIN from your terminal. In fact, we'll show you how in this section. However, this command is usually run automatically by another OS-9 utility command, TSMON. TSMON, OS-9 shorthand for timesharing monitor, makes multi-user operation possible on OS-9 computers.

When you — or TSMON — first run LOGIN, it requests a user name and a password. You answer its questions first. Then, it checks your answers against the information stored in a password file stored in the SYS directory on device /d0. You have three chances to answer each question correctly. If you don't, LOGIN aborts and sends you back to the OS-9 Shell.

Using information from your password file, LOGIN sets up your user number, working execution directory and working data directory. It then runs a program you have named in the password file.

Here's some more shorthand. If you answer LOGIN's prompt for a user name with <ENTER> you will be logged on as user number zero. This means you'll be the "superuser." Do this until you build your own secret file.

Go ahead, give LOGIN a try!

OS9: login <ENTER>

OS-9 Level 1 Timesharing System Version 1.2 84/09/01 21:25:28
User Name? michele <ENTER>
Password? tiffy <ENTER>>

Process #04 logged 84/10/07 21:26:05
Hello Michele! Welcome to OS-9

SETIME

The Color Computer appears to keep time when you run OS-9. Unfortunately, it does not contain a real clock and must be set each time you start OS-9. You must set the time with the OS-9 SETIME utility command.

SETIME does more than set the date and time however. It also starts the OS-9 clock module. If the OS-9 "Clock" is not running you cannot run more than one program at a time.

If you have a hardware clock chip with battery backup in your computer, you need only give the year when you run SETIME.

SETIME will get the rest of the information it needs from the clock chip.

You may type the date and time on your command line when you run SETIME. Or, you may wait for it to ask you for the date and time.

Here's one command line that will work.

OS9: setime 84,10,09,2143 <ENTER>

Notice that we typed commas between the year, month, day and time. You could just as easily have typed spaces.

OS9: setime 84 10 09 2151

Or, you could have run the date and time values together like this.

OS9: setime 841011 214535 <ENTER>

If you use this command line, make sure that you put the space between the day and the time. Here's the line to use if you have a hardware clock.

OS9: setime 85 <ENTER>

If you are skeptical and don't believe that the SETIME utility command works, run the OS-9 DATE utility to check.

SLEEP

Computers need sleep, too! No, that's not the real reason OS-9 has a SLEEP command, but maybe we woke you up with that one. You need to be awake to follow this!

SLEEP is another OS-9 utility command that does almost exactly what its name implies. It puts a process to SLEEP. How does OS-9 know how long to sleep? You tell it, by typing the number of ticks you want it to sleep.

You'll need to test the SLEEP utility on your computer to find out how many ticks it takes to add up to a second. On most OS-9 Level One computers, a tick is about 100 milliseconds long. Most OS-9 Level Two computers use a 10 millisecond tick. One tick is 16.66 milliseconds long on the Color Computer.

So what do you do with SLEEP? First, its a good way to stall and can cause quite a time delay. Or, you may want to use it to make the computer share the CPU when it is running several tasks that are time intensive.

Here's the SLEEP command line.

OS9: sleep 10 <ENTER>

If you type this command on a Color Computer running OS-9, the Shell will go to sleep for 166.6 milliseconds — or .166 seconds.

You can also get into trouble with the SLEEP command if you aren't careful. If you run the SLEEP command and type a count of zero, you will put your computer to SLEEP indefinitely. Unless there is an error, you will need to re-boot OS-9 to use your computer.

TMODE

We promised earlier that you could change the system if you didn't like it. Our next utility command will do just that. It lets you change 15 of your terminal's characteristics.

Microware wrote the TMODE command to give you a way to change the way your terminal responds. It reads the characteristics you type in a command line and installs them in the device driver of the path you are changing.

TMODE makes its changes on the device that is connected to the standard input path when it is run. It is used most often with your terminal. On the Color Computer the standard input path is usually connected to the keyboard.

You can make TMODE work on any path by typing a period. followed by a path number on the command line. And, you can change many things with TMODE. For example:

TO MAKE YOUR TERMINAL	TYPE THIS
Print only UPPER case letters	upc
Print both Upper and lowercase letters	-upc
Erase a character with the Backspace key	bsb
Backup but not erase a character	-bsb
Backspace over a line when you delete it	bsl
Send a line feed when you delete a line	-bsl
Echo all characters input to screen	echo
Do not echo characters	-echo
Send a linefeed with each <RETURN>	lf
Send a <RETURN>, but no linefeed	-lf
Stop listing when screen is full	pause
Scroll continuously	-pause
Send <decimal number> nulls at the	

end of each line	null=<decimal number>
Set video page length to "number" lines	pag=<decimal number>
Define backspace character	bsp=<hexadecimal number>
Define backspace echo character	bse=<hexadecimal number>
Define character that deletes a line	del=<hexadecimal number>
Define the bell character	bell=<hexadecimal number>
Define the end-of-file character	eof=<hexadecimal number>
Define end-of-record character	eor=<hexadecimal number>
Initialize ACIA status byte (Note: Since the Color Computer does not use an ACIA, it does not use this TMODE parameter)	type =<hexadecimal number>
Define character that reprints a line	reprint=<hexadecimal number>
Define character that returns last line	dup= <hexadecimal number>
Define character used to pause scroll	psc= <hexadecimal number>
Define OS-9 abort character	abort= <hexadecimal number>
Define character used to stop program	quit= <hexadecimal number>
Define X-ON character	xon= <hexadecimal number>
Define X-OFF character	xoff= <hexadecimal number>

Remember, if you are using a Color Computer, you cannot set the TYPE parameter with TMODE.

It's time to go to work!

OS9: tmode -upc -lf null=0 pause <ENTER>

This command line tells OS-9 that you want your terminal to print both upper- and lowercase letters but you don't want it to echo a linefeed after each carriage return. Additionally, it lets OS-9 know not to send any nulls at the end of a line and that it must pause after it fills each screen page.

OS9: tmode page=12 pause bsl bsp=8 <ENTER>

This command line tells OS-9 that the terminal screen is only 12 lines deep, that it should stop after each page is filled, that it should delete a line by backspacing over it and that the backspace character on the terminal has a value of 8 — the standard ASCII backspace.

If you plan to use TMODE to change the way your terminal operates when you start OS-9, read the next paragraph carefully!

When you use TMODE within a procedure file you must tell it which path you want to change. Since you are writing to the the standard output when you start OS-9, you need to change the device descriptor connected to path #1. To do this you must use the following command line.

OS9: tmode .1 -upc

Note that you must use the numeral one, not a lowercase letter 'l' following the period in your command line.

If you run TMODE, but do not tell it what you want to do, it will tell you how the device attached to your standard input path is configured.

TSMON

We've already shown you how to sign on to an OS-9 system from a remote terminal, so there's not much more we can say about the TSMON utility command. But, let's review.

TSMON must be one of the most patient programs in the OS-9 system. Why? Because it doesn't do anything but wait.

What does TSMON wait for? For you — or someone else signing on a terminal connected to your computer — to type a carriage return. When TSMON receives a carriage return it celebrates by bringing OS-9 to life on that terminal. Then it runs LOGIN, the OS-9 utility command we described earlier in this chapter.

If you are an authorized user and you haven't dropped a bit from your password, TSMON rewards you by assigning an OS-9 Shell to you for your own use.

When you have finished computing, you may log off your terminal by sending an end-of-file character as the first character of a command line. On most OS-9 based computers, you do this by typing the <ESCAPE> key.

To log off from a remote terminal connected to a Color Computer, you type the <CLEAR><<BREAK> combination.

Here's what your command line will look like.

OS9: tsmon /t1& <ENTER>

This command line starts your timesharing monitor on device /T1. Since you used the ampersand (&) at the end of the line, OS-9 will push TSMON into the background and return you to the Shell. As soon as you see the "OS9:" prompt you may continue to operate your terminal.

UNLINK

Remember when we told you that you must never forget to UNLINK a module after you complete a job? In this section we present a command you can use to do the job.

UNLINK gives you a way to manually tell OS-9 that you are through with a module. When you type UNLINK followed by a module name on a Shell command line, OS-9 decreases the LINK count of the module by one. If the count becomes zero, OS-9 will remove the module from memory. This will make it available for other users on your computer.

If someone else is using the module when you type the UNLINK command, OS-9 will still decrease the LINK count of the module by one when you UNLINK it. If you have previously LOADed the module, or if you have LINKed to it, everything will work as planned. The LINK count will not become zero, OS-9 will leave the module in memory and the other person's program will continue to run.

But, be careful! What do you think would happen if you told OS-9 to UNLINK a module that you have not LOADed — it just happens to be in memory because a friend is using it. You guessed it, the LINK count would go to zero, OS-9 would take the module from its directory and release the memory it had been using. Your friend's program would crash. Friends like that are not in great demand.

Here's something else you need to watch out for. When you find yourself in the middle of a software development session where you keep testing new versions of the same program, make sure that you actually UNLINK the module after you test it — before you LOAD a new version.

Why? Because sometimes when you run a series of tests on a program, you wind up LINKing to a module several times. Since UNLINK only decreases the link count by one each time you run it, you must manually repeat the UNLINK command several times until you are sure you have removed the module from memory.

One way to do this is to UNLINK a module over and over again until OS-9 reports that it cannot find the moduie. It may not be an elegant method, but it works. Besides if you use the magic <CONTROL><A> key, it's easy. Remember, the <CONTROL><A> key lets you repeat your last command line with two keystrokes. The other alternative is to buy one of the OS-9 utility packages that contains an UNLOAD command. Several third party software houses sell them.

Go ahead and give it a try!

OS9: unlink Spell Look <ENTER>

This command line will decrease the LINK count of modules named Spell and Look by one. If either of the LINK counts becomes zero, OS-9 will remove it from the module directory and release the memory it had been using.

Earlier in this chapter we showed you how to change the characteristics of your terminal by using the TMODE command. I'll bet you're wondering why there is another OS-9 utility command that does the same thing.

Actually, TMODE and XMODE are different. Changes made by TMODE are temporary, while changes made by XMODE are permanent. When you change a terminal's characteristics with TMODE your changes last only as long as the process that was running when they were made. TMODE is usually called from a Shell. When the Shell that was alive when you ran TMODE dies, your terminal characteristics will return to the way they were.

Here's an example. Suppose you are writing a letter using the DynaStar text editor. When you are about halfway through, you decide that you need to change one of the TMODE parameters.

To do this you go back to DynaStar's main menu and type 'S' for Shell. You enter your TMODE command and then go back to your letter. Everything is working fine.

Later, after you have finished with your letter and exited DynaStar, you LIST a file to your terminal. Something's wrong! The terminal isn't doing what you told it to do with the TMODE command.

What happened? When you exited DynaStar, you killed the process named DS — the process that was alive when you changed your terminal parameters. TMODE had made your changes in a path descriptor, not in the device descriptor. This means that when the process died, the path descriptors died. So did your changes.

XMODE makes more permanent changes by writing new information into the device descriptor you are changing. After you make a change with XMODE, any process that uses that device descriptor will be affected by your changes.

On the Color Computer a lot of programmers use XMODE to make a permanent change to their Baud rate.

Try it once!

OS9: xmode /T1 <ENTER>

Since you didn't give XMODE any parameters in this command line, it should list the present characteristics of a terminal named /T1.

Our next command line tells OS-9 to let the screen on a terminal named /T2 scroll continuously.

OS9: xmode /T2 -pause <ENTER>

You should be aware that when we say XMODE makes a permanent change, that change is only in effect until you turn off your computer. To make it totally permanent you need to save the modified device descriptor and run the OS9Gen command to create a new OS9Boot file after you run XMODE. Use OS9Gen to save the OS9Gen device descriptors with the new characteristics in a new OS9Boot file. After you do this, your changes will be in place every time you start your OS-9 computer.

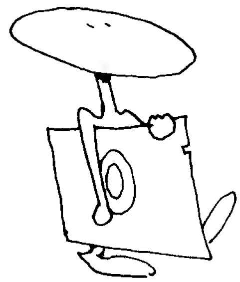
SUMMARY

In this chapter we've introduced you to a number of commands that let you change many of the operating characteristics of your computer. We complete our tour of the OS-9 utility command set in the next chapter with a look at a few Shell commands that are resident in memory. Join us there.

CHAPTER 15

shell commands that are resident in memory

OS-9 keeps a handful of Shell commands resident in memory at all times. Stop for a moment and consider why you would need these commands in memory, ready for use at a moment's notice.



Suppose that you change the disks you have mounted in your drives. If the command you need to change the current execution directory was not already in memory waiting for you to run it, how would you change your current execution directory? Well, that's why Microware leaves the CHD and CHX commands in memory.

In this chapter we'll show you eight commands that OS-9 leaves in memory all the time. You can type these commands at the beginning of a line or after any of the punctuation characters you use to separate OS-9 commands on a single line.

These characters include the semicolon which lets you run two utilities sequentially, the ampersand, which lets you run two commands at the same time and the exclamation point, which lets you build a "pipeline." You'll meet:

ex	t, -t
kill	w
p, -p	x, -x
setpr	*

EX

Usually when you run a program, an OS-9 Shell starts your program as a new process. The new process has the name you typed on your command line. When you start a process, the Shell

keeps some memory and gives some to the process. This happens because the Shell starts your program as a process and then waits around for it to die. If you run too many processes, you will run out of memory.

EX can help you solve this problem. It starts a process without a Shell. In the language of the OS-9 hacker, it "chains" rather than "forks" to the new process. When OS-9 chains to a process, the process that started that process disappears. This saves memory. Plus, EX is always in memory, ready to run.

There's something else you should know about EX. If you use it in a command line with more than one OS-9 command, it must be the last command on the line. Any commands that follow it on the line will never be run. Give EX a try!

OS9: ex tsmon /t1 <RETURN>

KILL

You can only use the <CONTROL><E> key — the <BREAK> key on the Color Computer — to stop a process that is running in the foreground. Occasionally, however, you will need to stop a process that is running in the background. The OS-9 Shell keeps the KILL command hanging around in memory so you can get the job done.

When you type KILL followed by a process number on an OS-9 command line, you effectively send an abort signal to the process with that process ID number. Before you can do this though, you must own that process. If you try to KILL a process that belongs to someone else — OS-9 can tell because it has a different user number — you will fail.

Here's how you use KILL.

OS9: kill 5 <RETURN>

Don't forget, you need to know the number of a process before you can KILL it. If you forget a process number, you can see a list of all processes that are running on your computer by running the OS-9 PROCS utility command.

SETPR

If you have an important program to run, you can speed it up by giving it a higher priority than other programs running on your computer. When you increase the priority of a program, you give it more CPU time.

You use the SETPR utility command to set the priority of a process — a priority that may be as low as one or as high as 255.

Here's how you use it.

OS9:setpr 4 200 <RETURN>

The sample command line gives process number four a high priority.

p, -p, t, -t, w, x, -x

We'll present the rest of the OS-9 resident Shell commands in a table.

IF YOU WANT THE SHELL TO:	TYPE THIS COMMAND
Wait until a process dies	w
Abort process after an error	x (OS-9 does this by default)
Ignore all errors	-x
Send "OS9:" prompt	p
Not send prompt	-p
Copy all input lines to output	t
Not copy input to output	-t (OS-9 comes this way)

Here's a trick you can try with the resident Shell command named 't'. One OS-9 vendor uses it to print a banner from his startup file.

```
OS9: echo
t
                              *************************
                              **   Dale L. Puckett   **
                              **       DaleSoft      **
                              **Dale City, VA 22193**
                              *************************
-t
setime 84 ; * start clock
date,t
load load
load utils1 ; * load most-used utilities
```

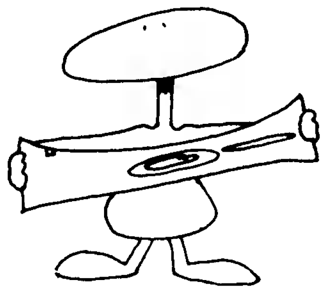
SUMMARY

In this chapter, we have introduced you to a few OS-9 commands that live in memory. It also completes our tour of the entire utility command set.

But, don't quit now. Join us in Chapter 16 for a look at several procedure files that use a number of the commands you have learned in the last six chapters. It will save you a lot of work.

CHAPTER 16

put os-9 to work with procedure files



In the last six chapters you have toured the entire OS-9 Utility Command Set. Now, we'll show you how you can combine these utility commands in procedure files which will really put OS-9 to work. You'll learn how you can use procedure files to:

Manage memory
Manage disk space
Manage system performance

Let's tackle the memory problem first. Since letting your computer do all the work for you is the name of the game, we'll write a procedure file to do the job. We'll call our procedure trimboot.

In order to show you what happens when you run a procedure like trimboot, we will make snapshots of several directories and files along the way.

First, let's set our goals and describe the job in English.

WHY CHANGE YOUR SYSTEM DISK

When you first use OS-9 you will most likely only take advantage of a very small part of its potential. For example, we can probably assume that:

1. You only own two disk drives
2. You don't own an external terminal

3. You don't own a printer
4. You don't understand filters, so you won't use pipes

Now let's look at the module directory of a Color Computer booted from a Radio Shack OS-9 system master disk.

```
Module Directory at 11:55:04
OS9       OS9p2     Init
Boot      CCDisk    D0
D1        D2        D3
CCIO      TERM      IOMan
RBF       SCF       SysGo
Clock     RS232     T1
PRINTER   P         PipeMan
Piper     Pipe      Shell
Mdir
```

Since memory space in a Level I OS-9 system is limited, it doesn't make sense to have modules in memory that we won't be using. If we make the assumptions above, we should be able to remove the following modules from our OS9Boot file.

```
D2        D3        RS232     T1
PRINTER   P         PipeMan   Piper
Pipe
```

The procedure file listed should do the job. Using commands in the OS-9 Utility Command Set, it instructs OS-9 to perform the following steps.

PROCEDURE FILE TRIM_BOOT

```
t
tmode .1 -pause
makdir /d1/MODULES
chd /d1/modules
load save
save CCIO CCIO
save CCDisk CCDisk
save IOMan IOMan
save SCF SCF
save RBF RBF
save SysGo SysGo
save Shell Shell
save Clock Clock
save TERM TERM
save D0 D0
save D1 D1
unlink save
OS9gen /d1 </d1/bootlist
deldir /D1/modules
d
del /D1/bootlist
```

```
chd /d0
dsave -s20 /d0 >/d1/makecopy
chd /d1
/d1/makecopy
del /D1/makecopy
tmode .1 pause
-t
```

Before you use trim_boot you must do the following:

1. Make a BACKUP of your Radio Shack system master disk by following the instructions we presented in Chapter Seven.

2. FORMAT a new disk and mount it in drive /D1.

3. Use the BUILD utility command to create a file named "bootlist" on the disk mounted in drive /D1.

4. Use the BUILD utility command to create a file named "trim_boot" on the disk in drive /D1. Trim_boot will contain the procedure file listed above.

   (NOTE: If you prefer, you may use the OS-9 EDIT utility command or a screen editor like DynaStar to build the files bootlist and trim_boot.)

The file, bootlist, should look like this:

```
CCIO
CCDisk
IOMAN
SCF
RBF
SysGo
Shell
Clock
TERM
D0
D1
```

If you study the list of filenames above, and the list of modules that the procedure trim_boot saves in the directory /d1/MODULES, you'll notice that they are identical.

The files you name in your bootlist file *must* always be available on a mounted disk, or the procedure trim_boot will fail. Further, OS-9 *must* be able to find them with the pathlist you use when you create your bootlist file.

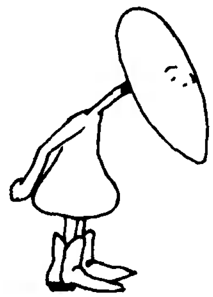
If you study the description of OS9Gen in the Radio Shack OS-9 Commands manual, you will learn that it creates, and links to, a new OS9Boot file. OS9Gen constructs the new OS9Boot file

from modules stored in a list of files it receives from the standard input path. You may even type that list on your keyboard. However, it is much safer to let OS9Gen read the list from a previously edited file.

After you have formatted your disk and installed the files bootlist and trim_boot, you can let your computer do the rest of the work. To get it started, type:

OS9: /D1/trimboot

HOW TRIM_BOOT WORKS



We'll follow the action play by play, so you will understand how procedure files work.

The procedure file trim_boot first uses the 't' Shell command to ask OS-9 to echo all input lines to the standard output path. Then, it uses TMODE to tell the system not to pause when the screen is full. Note here that you must always use the ".1" — numeral one, not lowercase 'L' — with the TMODE command when you use it within a procedure file.

Next, trim_boot tells OS-9 to create a directory named MODULES on the disk in device /D1. Then, it changes the current data directory to that directory. At this point, that directory is empty.

Next, trim_boot loads the OS-9 SAVE utility command and uses it to SAVE each module that you intend to place in your new OS9BOOT file. For example:

SAVE CCIO CCIO

This line tells OS-9 to save a module named CCIO in a file named CCIO in the current data directory. In the line before, trim_boot had instructed OS-9 to set the current data directory to /D1/MODULES, so the full pathlist to the new file is actually /D1/MODULES/CCIO.

Remember, when you run the SAVE command, the first parameter (name) is a pathlist to the file where you want to store the module(s) named on the remainder of the line. The modules that you are saving are all in memory. They were loaded from the original OS9Boot file when you booted OS-9.

After OS-9 has saved each of the modules you will need in your new OS9Boot file, trim_boot UNLINKs the SAVE utility command and runs the OS9Gen utility command, redirecting its input from the file, /D1/bootlist.

After OS9Gen creates the new OS9Boot file on the disk in device /D1 and links to it, trim_boot tells OS-9 to delete everything

in the directory /D1/MODULES and the file /D1/bootlist. They are no longer needed.

Let's look at a listing of the directory /D1/MODULES before it was used by trim_boot.

```
        directory of /D1/modules   12:34:24
CCIO      CCDisk    IOMAN     SCF
RBF       SysGo     Shell     Clock
TERM      D0        D1
```

Notice that all of those SAVE command lines in the procedure file trim_boot really worked. Before we continue let's look at a directory listing of the disk mounted in device /D1 after the line that DELetes the file named /D1/bootlist.

```
directory of/D1   12:34:06
OS9Boot   trim_boot
```

A file named OS9Boot has been stored on the disk in device /D1, and OS-9 has linked to it. OS9Boot contains the modules that were stored in each file listed in the file "bootlist." If you need to prove it to yourself, run the OS-9 IDENT utility command. Type:

OS9: Ident-s /d1/OS9Boot

After deleting the files in the directory /D1/MODULE, the directory itself and the file "bootlist," trim_boot tells OS-9 to change the current data directory to /D0. This drive should contain a backup of the original Radio Shack OS-9 system master disk.

HOW DSAVE WORKS

Next, trim_boot tells OS-9 to run the DSAVE utility command and save the results in a file called "makecopy" on the disk mounted in device /D1. It uses the -s20 option to tell OS-9 to take 20K of memory when it makes each copy. Let's list part of the procedure file "makecopy" so we can see how DSAVE works.

```
t
tmode .1 -pause
load copy
Makdir CMDS
Chd CMDS
Copy #20K /d0/CMDS/asm asm
Copy #20K /d0/CMDS/backup backup

etc...

Copy #20K /d0/CMDS/verify verify
Copy #20K /d0/CMDS/xmode xmode
Chd ..
Makdir SYS
```

```
Chd SYS
Copy #20K /d0/SYS/errmsg errmsg
Copy #20K /d0/SYS/password password
Copy #20K /d0/SYS/motd motd
Chd ..
Makdir DEFS
Chd DEFS
Copy #20K /d0/DEFS/OS9Defs OS9Defs
Copy #20K /d0/DEFS/RBFDefs RBFDefs
Copy #20K /d0/DEFS/SCFDefs SCFDefs
Copy #20K /d0/DEFS/SysType SysType
Chd ..
Copy #20K /d0/startup startup
unlink copy
tmode .1 pause
-t
```

After the DSAVE utility command creates the procedure file makecopy, trimboot tells OS-9 to change the current data directory to /D1. It then tells OS-9 to run makecopy.

Inspection of makecopy shows that it contains every command needed to make a new copy of the system disk in drive /D0 on a freshly-formatted disk mounted in device /D1.

All you need to do now is sit and watch. OS-9 will make all the directories it needs, and copy all the files DSAVE has told it to.

After OS-9 finishes running the procedure file makecopy, trim_boot tells it to delete makecopy. It then uses TMODE to restore the current standard output path to its original condition, turns off the echo of lines input to the SHELL and exits gracefully. You now have a brand new disk containing a new OS9Boot file that you designed.

Use these listings as examples when you design your own procedure files. They work, and work well. Once you type the name of a procedure file, OS-9 takes over and runs your computer for you.

A TRICK THAT SAVES TIME

Now, we'll look at a procedure file that takes a different approach.

Memory is not the only thing that is limited on the Color Computer. Since the Radio Shack disk drives are limited to 35 tracks on a single side of the disk, mass storage is also at a premium. And since these drives only step at 30 milliseconds per track, operation is slow.

You can improve performance some if you put a few of the utility commands that you use often in the OS9Boot file. This

means that OS9 won't need to load them from your /D0/CMDS directory before it runs them. It also means that you won't need to store a copy of them in this directory.

In this example, we will assume that you want to use your original OS9Boot file — or the one you just made following the example above — plus five new modules. We will add: DIR, DISPLAY, LIST, MDIR and MFREE.

Begin by using your present system disk to boot OS-9. Then, use the BUILD utility command, or your favorite editor, to enter the following procedure in a file named /D0/make_new_sys.

Also, enter a list of the modules you want in your new OS9Boot file into a second file named /D0/new_boot_list.

After BUILDing make_new_sys and new_boot_list, run the procedure make_new_sys by typing:

OS9: /D0/make_new_sys

THE PROCEDURE MAKE_NEW_SYS

```
t
tmode .1 -pause
format /d1 </term
makdir /d1/MODULES
chd /d1/modules
load save
save CCIO CCIO
save CCDISK CCDISK
save IOMAN IOMAN
save SCF SCF
save RBF RBF
save SysGo SysGo
save Shell Shell
save Clock Clock
save TERM TERM
save D0 D0
save D1 D1
unlink save
os9gen /D1 </D0/new_boot_list
deldir /d1/modules
d
del /d0/new_boot_list
chd /d0
dsave -s20 /D0 >/D1/make_copy
chd /d1
/d1/make_copy
del /d1/make_copy
tmode .1 pause
-t
```

THE FILE NEW_BOOT_LIST

Here is a snapshot of the file new_boot_list. Notice that since the modules, DIR, DISPLAY, LIST, MDIR and MFREE are already in files, stored in the directory /D0/CMDS, you won't need to save them in /D1/MODULES. OS9Gen can load them from the directory /D0/CMDS. However, you must give OS9Gen the complete pathlist to these files since your current data directory is set to /D1/MODULES when you run OS9Gen.

```
ccio
ccdisk
ioman
scf
rbf
sysgo
shell
clock
term
d0
d1
/d0/cmds/dir
/d0/cmds/display
/d0/cmds/list
/d0/cmds/mdir
/d0/cmds/mfree
```

Remember, each and every module that you want in your final OS9Boot file must be in a file that is named in the list of files named in new_boot_list. Conversely, all modules that you do not want in your OS9Boot file must be excluded from the list.

The procedure file make_new_sys will place a new OS9Boot file on the disk mounted in drive /d1. Each module stored in a file listed in the file /d0/new_boot_list will be in the new OS9Boot. OS9Gen gets the list of names from this file instead of the terminal because its input has been redirected from this file.

The cost of using this approach is billed in memory. The payoff is in speed. Name your poison. But remember, once these commands have been installed in the boot file, they are as good as in ROM (Read Only Memory). You'll never need them in your working execution directory again.

SOMETHING TO THINK ABOUT

In fact, it might be a good idea to create several different OS-9 system disks. One could bring the system alive with BASIC09 in memory and many packed BASIC09 intermediate code modules in the execution directory.

Another system disk could come alive with a word processor like DynaStar in memory and a number of word counting and file

handling utilities in the execution directory. Go to it. An OS-9 system can be set up just about any way you want it. Your imagination is almost the only limit.

OTHER WAYS TO GET MORE ROOM ON A DISK

As you have probably discovered by now, it doesn't take long to fill up the 620 sectors available on a disk used with a Radio Shack single-sided, double-density drive. Let's look at another way to tackle the problem.

Back when you could only buy single-sided, single-density drives, you had to force yourself to get organized.

I used two drives. The first drive was always the system drive. The second was always the work drive. I used the first drive to hold all command files. I used the second to hold all text files or other data.

I used one system disk when I was programming. It contained an editor, an assembler, a debugger, a BASIC interpreter and several small utilities that I needed all the time. Programs like DIR and LIST fall in this category.

I used another system disk for word processing. It held an editor and a text formatter, plus several handy word processing utilities. The most-used commands, like DIR and LIST, were stored on this disk along with several small programs that counted words, checked readability, etc.

On the work drive, I used different disks for different jobs. One disk held only BASIC programs, another, magazine articles, and yet another held college homework. It took a lot of disks, but I almost always had enough room on the disk to get the job done.

Now, let's see how our philosophy can be applied to OS-9 on the Color Computer. Here's one way to implement this strategy.

First, write protect your original system disk and use BACKUP to copy it to a freshly formatted disk. Then, we'll change your plain vanilla OS-9 system disk to a BASIC09 system disk.

Start by printing a listing of the programs in the CMDS directory. Then, consider how often you would use each of them while running BASIC09.

When running BASIC09, it is highly unlikely that you will ever need to use a utility command like EXBIN. Study the directory listing and see how many of the utility commands you can get along without.

The list will most likely include asm, backup, cmp, cobbler, dcheck, debug, deldir, dsave, dump, edit, exbin, format, ident,

os9gen, shell and quite a few more of your own choosing. Here's a sample procedure file that will do the job. You'll have to make the decision about which files to delete yourself.

```
t
tmode .1 -pause
chx /d0/cmds
chd /d0/cmds
load del
del asm backup cmp cobbler dcheck
del debug dsave dump edit format ident
del login pwd pxd os9gen shell tsmon
del save sleep setime verify xmode
unlink del
tmode .1 pause
-t
```

Remember, you may need copy, date, del, deldir, dir, display, echo, free, link, list, load, mdir, mfree, rename, tmode and unlink, so leave them on the disk.

Since you are creating a BASIC09 system disk and won't be using it for any assembly language programming, you can also delete the entire DEFS directory. This directory contains assembly language code that defines all OS-9 system equates.

To delete this directory, type:

OS9: deldir DEFS

Answer the prompt that appears with a ' d ' for delete, and in a few minutes you will have a lot of free space on your BASIC09 system disk.

OTHER PROCEDURES THAT IMPROVE THE SYSTEM

If you own a 40-track drive, you can modify the device descriptor so that OS-9 will use the entire disk. Remember, as shipped, Radio Shack OS-9 only works with 35-track drives.

Here's a way to modify the device descriptor module /D0 so that you can use the last five tracks of a 40-track drive. If you have two 40-track drives installed, don't forget to change both drive descriptors. The keyboard sequence looks like this.

```
OS9: debug <ENTER>
DB: l d0
     C10B 87
DB: . <SPACE> .+18
     C123 23
DB: =28
     C124 01
DB: q
```

After you change this byte in device descriptor /D0, OS9 will know that you have 40 tracks available on that device. The five new tracks will be available on any disks you format after making this change.

If you don't want to make these changes with the DEBUG utility command by hand, you can use this procedure file. Put it in a file named /D0/To_40_Track.

```
l d0
. <SPACEBAR> .+18
=28
q
```

Use the BUILD utility command to create the file. Then run your new procedure file by typing:

OS9: debug </D0/To_40_Track <ENTER>

This command line is fascinating to the OS-9 novice because it shows how a program as complicated as DEBUG can be run from a file of pre-edited instructions. It's amazing to watch.

Once you have changed the device descriptor, you may use the OS9GEN command to create a disk that will come on line with 40 track drives for /d0 and /d1. Remember, you must save the modified device descriptor in a temporary file — you could call it D0.new — and run the VERIFY utility command with the update option turned on before you can put the new module in your OS9Boot file.

This command line sequence will solve the problem and update the CRC for you.

```
OS9: save d0.new d0
OS9: verify <d0.new >d0 u
OS9: OS9Gen /D1 <filelist
```

All modules you want to include in your new OS9Boot file, including the newly verified file d0, must be included in the filelist file.

USING DRIVES WITH FASTER STEP RATES

Here's a way to let OS-9 take advantage of disk drives that can step from track to track at a faster rate. It modifies CCDisk each time you boot your system.

Start by putting this line into your file /d0/startup file.

debug </d0/changedisks

This command line will load the debugger from your system

directory and execute it. Instead of getting its instructions from you on the keyboard, however, debug will get them from a file named "changedisks" on device /d0.

Before creating the procedure file /d0/changedisks, study this table.

TABLE OF CHANGES TO MODULE CCDISK
TO CREATE FASTER STEPPING RATE

OFFSET INTO MODULE	OLD VALUE	FOR 6 MS	FOR 12 MS	FOR 20 MS
01FE	13	10	11	12
0204-0205	2225	088B	088B	088B
02DD	43	40	41	42
02E9	03	00	01	02

CHANGES TO MODULE CCDISK THAT MAKE SECOND SIDE
OF DISK MOUNTED IN A DOUBLE SIDED DRIVE IN /D0
APPEAR TO BE DEVICE /D2

OFFSET INTO MODULE	OLD VALUE	NEW VALUE
0210	04	41
0211	40	42

In the procedure changedisks, we used a value that will create a 20 millisecond stepping rate since most Radio Shack drives will step at that speed. If you have faster drives, you can use the values from the table above. The tables show you the value for stepping rates of both 12 and six milliseconds.

THE PROCEDURE CHANGEDISKS

```
$load echo
$echo Changing step rate in CCDISK
$echo to 20 milliseconds.
l ccdisk
. <SPACE> .+1fe
=12
l ccdisk
. <SPACE> .+204
=08
=8B
l ccdisk
. <SPACE> .+2dd
=42
l ccdisk
. <SPACE> .+2e9
=02
$echo Stepping rate has been changed
$unlink echo
q
```

You could go one step farther and let the debugger tell

CCDisk that your drives can use 40 tracks instead of 35. To do that you would just add the code we gave you earlier in this chapter to the bottom of the file.

It would also also be possible to use separate debug command files to change the step rate and change the number of tracks. Perhaps you could call them changestep and changetracks.

Then, you could run them both by putting the following sequence in your startup file.

```
load debug
debug </d0/changestep
debug </d0/changetracks
unlink debug
```

If you only need to make the changes part of the time, you can save the sequence above in a separate procedure file and then run it from the OS-9 command line when you need to make the changes.

Stretching the imagination a bit farther, you could use the procedure above to make these changes permanent — if you run it and then install your changes in a new OS9Boot file.

A procedure file like this would work.

```
chx /d0/cmds
load debug
debug </d0/changestep
debug </d0/changetracks
unlink debug
format /d1 </term
cobbler /d1
verify </d1/os9boot >/d1/boot.temp u
del /d1/os9boot
copy /d1/boot.temp /d1/os9boot
echo Disk in drive /d1 now has
echo a new CCDISK module that
echo will step at 20 milliseconds.
unlink echo
```

You will need to put a backup of your original Radio Shack OS-9 system master disk in drive /d0 before you run this procedure. Also, the file /d0/changedisks must be present on the disk mounted in drive /d0.

Do you remember how to add another line or two to this procedure file so that it will tell OS-9 to copy all the files on the disk in drive /d0 to the disk in /d1 after it changes the OS9Boot file? If not, you should review the first part of this chapter for the answer.

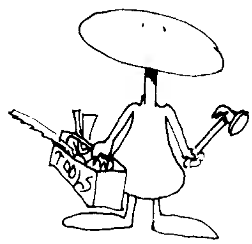
SUMMARY

In this chapter you learned how to put OS-9 to work with procedure files. If you turn your imagination loose, you will probably dream up hundreds of routine jobs that you can do with procedure files.

In Chapter 17 we introduce you to filters. As soon as you combine filters with pipes, you will begin to realize the real power of multitasking.

CHAPTER 17

just plain tools — the toolbox philosophy



Since OS-9 is modeled after the UNIX operating system, it seems like we would be way ahead in the game if we learned to work like UNIX programmers.

In this chapter we'll introduce you to five "toolkits." Almost all of the utility commands in these packages work as filters in OS-9 pipelines. Before we're through, you'll meet:

Microware's OS-9 File Handler Toolbox
D. P. Johnson's Hackers Kit
Computerware's Textools
FHL's Utilix and UniCharger

To tune into the toolbox philosophy, we need to learn to think of individual programs and utilities as tools. OS-9 lets us run more than one process at a time. That's half the battle. Now all we need to do is learn how to use several small tools together to do a big job.

Let's begin with a few examples. When I first ran OS-9, I did everything the hard way. I typed out every pathlist. I just didn't trust the machine. I was as non-productive as one could be. One of my typical command lines might have looked like this:

OS9:/d0/cmds/copy #16K /d0/cmds/greatbigprogram /d1/cmds/greatbigprogram

Then, I learned about the default directories. Life was much simpler:

OS9: chd /d0/cmds

OS9: copy #16K program /d1/cmds/program

That was much better. But then, every once in awhile I needed to copy a directory that contained 30 or 40 files. I used the magic key a lot. This saved about half the typing, but it was still a hassle.

Six months later, the OS-9 DSAVE utility command was released. It helped a lot when you needed to copy an entire directory — or even an entire disk. The commands went something like this:

OS9: chd /d1/directorytocopy
OS9: dsave /d1 >/d0/copy_procedure_file
OS9: chd /d0
OS9: /d0/copy_procedure_file

When I ran this sequence, the machine did most of the typing. DSAVE generated a file with a series of lines that looked similar to the first example above. At that time, I redirected them into a file and then ran it as a procedure file. It was really slick to watch the computer do all the work by itself.

Then, the process became another order of magnitude easier when OS-9 pipes were released about a year later. How would you like to trade the four lines of typing above for two command lines — and the first line really doesn't count. Write these Shell Command lines on a label and stick them on your keyboard. It will save you hours.

OS-9: Chd /d1/directorytocopy
OS-9: dsave /d1 ! (-x chd /d0/directory_to_copy_to)

When you run this command line, you'll be using your first OS-9 "pipe." The magic is in the exclamation point — the Shell command line symbol for an OS-9 pipe. Here's what happens when you run the command lines above.

The first line sets your current data directory to /d1/directorytocopy. DSAVE always saves the current data directory. In the second line, instead of redirecting DSAVE's output, we used a pipe, '!'.

OS-9's Shell sends the output of DSAVE to the standard output device, normally your terminal. But, since you typed the pipe symbol after the DSAVE command, the Shell pipelined the data straight into the command following the exclamation point, '!'.

In the second part of the command line, we tell the Shell not to abort on an error and to change the current data directory to /d0/directory_to_copy_to. After we do this, the Shell accepts each line from DSAVE just as if it were coming from the keyboard.

Each time the Shell finds a carriage return, it executes the

commands on that line. When each of the lines generated by DSAVE has been executed, your new directory will be ready to use.

MICROWARE'S FILE HANDLER TOOL BOX

Microware designed their toolbox so that most of the utilities could be used as filters. They read data from the standard input path, modify it in some manner, and then send it to the standard output device. Additionally, several of the programs in the package can take a list of filenames from your keyboard and execute the same command with each of them.

The ideas for this package came from the book "Software Tools" by Brian W. Kernighan and P. L. Plauger. All of the tools in the package are popular with UNIX programmers. The Microware Tool Box contains:

NAME	FUNCTION
Code	Displays the hexadecimal equivalent of a key
Count	Counts characters, words, and lines
Compress	Compresses a text file
D	Prints directory listing
Expand	Expands a compressed file
Grep	Globally Finds Regular Expressions and Prints them
Pr	Prints a file with formatting
Qsort	Performs Quick in-memory sort
Space	Spaces and/or indents a file
Split	Splits a file into several files
Tr	Transliterates a file's contents
Xmode	Examines or changes a device descriptor

Let's take a closer look at this utility package and show you how you can use several of these small tools on the same command line to do a big job.

When your directories get long — they really shouldn't with OS-9's hierarchical directory system — it can be a real hassle to look through a list of filenames in random order. Life would be much easier if you could look at a sorted list. Try this command line:

OS9: d ! qsort ! pr >/p

Now, imagine that you would like to know how many ' C ' source files you have in a directory. Let's set up a pipe to count them:

OS9: d *.c ! count -l

This command line should do the job if your ' C ' source files are in your current data directory.

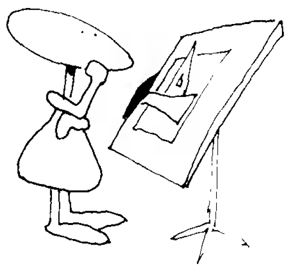
Imagine that you are an author and that you have just finished writing another chapter in the great American novel. You have full confidence in your abilities as a writer, but you realize you have a bad habit — you keep using tacky words.

Instead of writing "use," you write "utilize" — almost every time. If you want to make a quick check to make sure you don't slip up again, set up a pipe like this.

OS9: grep utilize Great.American.Novel.Chapt4 ! count -l

In a few seconds you'll know how many times you used the word utilize. With a full toolbox you can find out just about anything you would ever want to know. In fact, you could even use another tool from this kit to change every occurrence of "utilize" to "use."

NOW WHAT, CONTINUED



Here's another example of what you can do when you understand filters, pipes and apply a little imagination. This filter does a real job.

How many times have you wanted a simple data base manager that would let you look up a phone number fast. Try this:

OS9: BUILD phone
? dick herring, 515-555-1212, des moines, ia
? lonnie falk, 502-228-4492, prospect, ky
? jim reed, 502-228-4492, prospect, ky
? <ENTER>

This sequence uses the BUILD utility to place three names with the proper telephone number, city and state, in an OS-9 text file.

Now, imagine that you answer the phone and someone asks you for Lonnie's number. The stack of little yellow telephone slips on your desk is more than a foot high and someone has taken his card from your card file. What can you do?

If you have a Color Computer running OS-9 sitting on your desk and you have installed one of the UNIX like toolboxes, you can type:

OS9: grep Lonnie phone

In a second or two this line will appear on your screen:

lonnie falk, 502-228-4492, prospect, ky

Let's try another example. What can you do if you remember a long lost friend in Kentucky from your ham radio days, but you

can't recall his name. Try this:

OS9: grep ky phone

Seconds later you'll see this message on your screen.

lonnie falk, 502-228-4492, prospect, ky
jim reed, 502-228-4492, prospect, ky

How's that for an easy and cheap electronic phone book?

MICROWARE'S TOOL KIT HAS SEVERAL PLUSES

Three of the tools in the Microware File Handler Tool Box deserve special mention.

First, I almost always use the ' D ' utility instead of the standard OS-9 DIR command now. This tool is powerful because of its wildcards.

b* matches BCD, badfile, b, bird.c or b.tempfile
***c* matches file.c, BCD, or account.bak**

***f matches file.f, testf, temp.f or photograf**
***.src matches spell.src, lookup.src, LIST.SRC or any.src**

Study the matches in the table above and you'll see the power of this utility. Of course, since it outputs one filename on each line, it is very easy to use in a pipeline.

Two tools from the File Handlers Tool Box that really shine are GREP and TR. They are powerful because they recognize a set of regular expressions. Included in the set are operators that match any ASCII character, closure, a character class, an EOL character and a special escape symbol. All the standard escape symbols are recognized, including:

\t = tab character
\n = new line character
\b = backspace character
\f = form feed character

GREP and TR will both recognize any ASCII numeric value between 1 and 127 that immediately follows a backslash. For example, \32 will cause GREP and TR to recognize an ASCII space.

TR is an especially useful tool because it gives you a way to go through a file and convert any string of characters to any other string. To get a feel for its power and shorthand, study this command line.

OS9: TR [a-z] [A-Z] myfile
OS9: TR [A-Z] [a-z] myfile

If you run these two command lines you will see that they are equivalent to the UNIX commands UPPER and LOWER. In the first command line TR converts any character between ' a ' and ' z ' to the corresponding capital letter.

The second command does just the opposite. Since TR recognizes the regular expressions we mentioned earlier, it can be used to change just about anything to just about anything else.

FILTERS, FILTERS AND MORE FILTERS

Several other Third Party Software developers have followed in Microware's steps and released Unix-like filter kits. We'll look at D. P. Johnson's tool kits next.

FILTER KIT #1

NAME	FUNCTION
LS	lists filenames, one per line
BUF	buffers input until eof, then outputs it
CP	copies files from working directory
DL	deletes a list of files from current directory
FLIST	lists files to standard output
INFO	displays owner, creation date, attributes of a file
MV	moves files from directory to directory without copying
PAG	formats data into pages
REMOVE	removes a file from data directory without deleting
SELL	changes owner number of named files
SETAT	resets file attributes of list of files
SORT	sorts list of up to 300 names

HACKER KIT #1

DISINP	disassembles code from standard input
FILTER	strips all occurrences of a character from standard input
MEMLIST	lists memory in unformatted binary
MEMLOAD	loads files into memory at absolute address
REWRITE	writes standard input into a file at specific offset
SPLIT	splits data from input file into one or more output files

Johnson's utilities use only the standard input and standard output paths. This means their input and output can be redirected to various devices or files, or piped to other processes. Utilities like these are meant to be used together.

For example, LS outputs a list containing the names of files stored in the current data directory, one on a line. This lets other commands in the tool kit perform an operation on a list of files.

DL, deletes all files named. PAG paginates a listing of the data contained in the files named. SORT sorts any data it receives from

the files named on the standard input path. These command lines should make things clear:

OS9: ls
OS9: ls ! dl
OS9: ls ! pag
OS9: ls ! sort ! pag

The first command line lists the names of files stored in your current data directory, one name per line. The second deletes all files in your current data directory. The third, prints a paginated listing of filenames from your current data directory. And, the last command line prints a sorted and paginated listing of the names in that directory.

Normally, you wouldn't want to delete all the files in a directory. But, imagine for a minute that you need to delete all files that begin with the letter 'a'. To do this, you could use the following command:

OS9: ls a* ! dl

If you only want to delete files that begin with the letter ' a ' and have filenames three characters long, you could type:

OS9: ls a?? ! dl

Or, perhaps you need a list of the names of all directory files in a directory. This command line should do the job:

OS9: ls -A.d.

Here, LS checks to see that the directory attribute is on before it lists the filename. LS also lets you check for attributes that are off, the owners number, and the date. There is even an option that lets you list the names of files that were created before a certain date.

Johnson's CP utility lets you copy files with the filenames listed on the standard input path from the current data directory to a file with the same name on the destination path.

OS9: chd /d1/SOURCE
OS9: ls -nby83m7 ! cp /d0/SOURCE.BAK

These command lines will copy files in the directory /d1/SOURCE into the directory /d0/SOURCE.BAK if they were not modified before July of 1983.

Here are a few more examples of D. P. Johnson's utilities in action. Using INFO is similar to running the standard OS-9 utility command DIR with an ' e ' option. It displays the owner number, the creation date, modification date and time, attributes, byte-

count and the name of a file. Here are two forms of its use:

OS9: info ls cp dl
OS9: ls -e ! info

MV is a handy utility because it lets you move a file from one directory to another without actually copying the file. Needless to say, this is much faster. The two directories must be on the same physical device however.

Suppose you have all the commands you want in a directory, but they are in random order. You would rather have them filed in sorted order. Try this sequence of commands:

OS9: makdir /d0/CMDS.SORTED
OS9: chd /d0/cmds
OS9: ls ! sort ! mv -l /d0/cmds.sorted

D. P. Johnson's HACKER's KIT #1 does not contain filters. However, it gives you a set of tools that are very useful to anyone working with assembly language programming or customizing their operating system.

The disassembler that comes in the package does not receive its input from a file as do most disassemblers. Rather, it reads its input directly from OS-9's standard input path. This opens up many possibilities, and you can use it to disassemble code that is stored in memory or disk files.

Johnson's FILTER KIT #2, which was released just as this book went to press, contains another 10 OS-9 utilities. The most notable is MACGEN — short for Macro Generator. It builds a module out of a list of Shell commands that can be run with one command. The function it performs is similar to that of a Shell procedure file. However, it is faster, and allows you to use parameter substitution in your command line.

Utilities in this release include: Append, Confirm, FF, ForcError, MacGen, NulDevice, Rep, Size, Touch and UnLoad. The Rep command lets you repeat programs designed for use with a single parameter by feeding it a list of names on the standard input path.

UTILIX — UNIX-LIKE TOOLS FROM FHL

UTILIX brings UNIX-like utilities to OS-9. It is a set of tools from the Frank Hogg Laboratory in Syracuse, NY.

UTILIX

CAT	concatenates text files and lists to standard output
CODE	prints decimal and hex values of character typed
CRYPT	encrypts files

DIFF	compares two files line by line and reports differences
DISPLAY	displays the ascii, decimal, hex or octal value typed
GREP	searches file for expression
LOWER/UPPER	converts all characters to lowercase or uppercase
PACK/UNPACK	compresses and decompresses text files
PR	lists and formats files to standard output
SORT	sorts a file with up to 10 keys
TAIL	prints the last part of a file
TIME	times the execution of a command
WC	counts characters, lines and words

Most of the tools in this kit use standard UNIX command names and their command line syntax is for the most part identical to the corresponding UNIX command.

CAT for example, lists text files to the standard output path. This doesn't sound like much, but let's see what happens when we apply a little creativity.

The most obvious use for CAT is to merge a number of files into one.

OS9: cat file1 file2 file3 >bigfile

The redirection operator at the left end of the command line directs the standard output path into a file named bigfile located in your current data directory.

Now, let's show you a trick with CAT. What do you think this command line will do?

OS9: cat >workfile

Believe it or not, it emulates the standard OS-9 BUILD utility.

The CODE utility prints both the decimal and hexadecimal value of the character you type. If you like to keep secrets, you'll find CRYPT an interesting tool. You supply the secret code word and it ciphers the file.

OS9: crypt mycode <myfile >secretfile
OS9: crypt mycode <secretfile

The first command line ciphers the text in a file named "myfile" located in the current data directory, using the code word, "mycode." The second, deciphers that file and lists the original English language text on your terminal.

DIFF is one of the slickest programs in the UTILIX package. It compares two files on a line by line basis. If lines are missing from a file, it tells you which lines are missing and where they should be located. If a file has extra lines, it finds them for you. If the two files have different lines in them, DIFF will tell you which lines to replace to eliminate the differences.

The DISPLAY utility is similar to the standard OS-9 Display utility, except it is more versatile. The original DISPLAY will only display hex values to the standard output path. This DISPLAY will take ascii text, as well as decimal, hexadecimal and octal input, and display the result on your screen or printer.

The UTILIX SORT utility is not limited to data in memory, so it can sort fairly large files. You are limited only by the amount of disk space you have available for work files.

TAIL is very handy and also versatile. It lets you look at the last few bytes, characters or lines of a file. You tell it how many characters or lines you want to see. For example:

OS9: tail -10l myfile
OS9: tail -10c myfile

The first command line will let you see the last 10 lines of "myfile." The second will only let you see the last 10 characters.

TIME is a useful program when you are benchmarking a procedure. Want to know how long it takes to list a file? Ask your trusty Color Computer.

OS9: time list testfile

And finally, WC is a utility that lets you count the number of characters, lines or words in a file. If you don't tell WC what you want to count, it will count all three.

UNIX LOOK OUT!

One programmer came up with a creative strategy recently that should help everyone using OS-9. Brian Lantz examined each OS-9 toolkit. Then, he studied the standard UNIX utilities.

Lantz didn't want to waste his time reinventing the wheel, so he struck out to break new ground. His goal was to fill the gap between OS-9 and UNIX.

His first package contains 17 UNIX-like utilities and is being marketed by Computerware in Encinitas, CA.

COMPUTERWARE'S TEXTOOLS

CAT	—	concatenates standard input to standard output
FGREP	—	a fast, GREP-like pattern matching utility
LOWER	—	converts all uppercase letters to lowercase
LS	—	displays list of file names in a directory
PACK	—	compresses multiple spaces in a file
PR	—	formats output to standard output path
QSORT	—	sorts data in memory and sends to standard output
RPL	—	replaces "key1" from standard input to "key2"

SPLIT	—	splits source file into separate files
TAIL	—	displays last 10 lines of files
TIME	—	tells how long it takes to execute a command
TR	—	translates characters for standard input
UNIQ	—	removes duplicate lines from standard input
UNPACK	—	opposite of PACK
UPPER	—	converts all lowercase letters to uppercase
UPS	—	repeats command using filenames from standard input
WC	—	counts characters, words, and lines in standard input

The Computerware package is probably the best investment for Color Computer users. The reason? Each utility in the Textools set is written entirely in 6809 assembly language. This means the code is short and runs fast.

The Textools set is almost identical in function to the FHL UTILIX package. However, the tools in UTILIX were written in C and are five to 10 times longer. In an OS-9 Level I system where memory is at a premium, Textools appear to be the best choice.

Wherever possible, the Textools obey all UNIX conventions. This means they use the same command line and work just like their UNIX counterpart. Like the Microware GREP and TR utilities, most of the Textools recognize meta characters and regular expressions.

In fact, Lantz has added a utility called META that pre-processes meta characters for commands that don't recognize them. For example, study these command lines:

OS9: meta del test* <ENTER>
OS9: meta list /d0/LISTINGS/e* <ENTER>

UNICHARGER MAKES OS-9 BEHAVE LIKE UNIX

In addition to the Textools utility set, Lantz has written a package called the "OS-9 UniCharger" for Frank Hogg Laboratory. This package, designed to make OS-9 behave more like UNIX, is divided into two sections.

UniCharger SYSTEM COMMANDS

AT	set up a procedure file AT a future date and time
ATRUN	run a procedure file set up by AT
CHECKMAIL	tells you when you have mail
CONFERENCE	stdin goes to selected users online
CRON	run tasks periodically
FINGER	find information about users on line
LOGIN	similar to Microware's, checks for mail
MAIL	Send and receive mail from user to user
MAN	Online system manual
PASSWD	change and encrypt login password
SU	Switch Users without LOGIN
WALL	write file to all users on system

WHO	tells who is on system and where they are logged on
WRITE	write message to a specific user
PROFILE	an information file used by many of these programs

UniCharger UTILITY COMMANDS

CAL	print a calendar of any year
CHMOD	changes attributes of file or directory
CHOWN	changes ownership of a file
COMM	compares two files
CRYPT	encrypt standard input to standard output
DU	check usage of disk space
ERROR	list error message
EXPAND	add input from stdin to a "pathname"
HEAD	print first few lines of file
META	expands meta characters from comand line
MV	movesfile
TEE	UNIX equivalent, overwrites existing file
TTY	Display name of standard input device
UDATE	display current date and time, UNIX format
VIS	display non printing characters as \nnn

The "System" utilities above must be installed on your system. They change a few of the OS-9 system files and add features that make OS-9 look very much like UNIX.

For example, the structure of the OS-9 password file is changed so that a PASSWD utility command can be used to let individual users change their passwords. On standard OS-9 systems, only the superuser can change the password file. The new LOGIN command checks for MAIL.

The UniCharger utility commands stand alone. They are all patterned after their UNIX counterparts and were written to bring more UNIX-like functions to OS-9 programmers.

HIRES GIVES YOU A 51 COLUMN SCREEN

In addition to the filter kits we have highlighted so far, we need to mention several other software packages that greatly enhance OS-9 operation on the Color Computer. The first deals with the problem of the 32 column screen.

HiRes lets you write high-resolution text on an OS-9 graphics screen. This means your programs can use the 24 by 51 screen of uppercase or lowercase characters, with descenders, and draw graphic elements on the same screen at the same time.

OS-9, as shipped by Tandy, uses a device driver called CCIO. It gives you graphics capabilities and lets you reserve memory for a graphics screen — like the PMODE 3 or PMODE 4 statements in Extended Color BASIC — and gives you a way to set or erase points, or draw lines and circles on the screen.

When you run HiRes, both the text and graphics are output through /TERM — OS-9's standard output device. HiRes draws the text characters using information from a separate "Character Set" module which can be modified by the user.

The information in the character set module includes the shape and size of each character, the spacing of each row and column, and the default graphics mode and color set used to draw the characters. The standard character set draws each character in a seven-dot by four-dot matrix with one-dot descenders. The descenders make the display easy to read. Characters appear black on a buff background.

All of the capabilities in the original OS-9 display module still exist while you are running HiRes. This is possible because HiRes passes the graphics commands on to CCIO. Other commands let you return to the 32 by 16 alpha display and de-allocate the graphics memory used by HiRes. You may want to do this when you are running long programs that require a lot of memory. The Radio Shack C compiler is one program where you must do this.

HiRes uses all of the CCIO commands that control the text cursor, including direct cursor addressing, clear screen, erase line and so on.

HiRes gives you additional features that enhance the text display. A good example of this is the "Erase from cursor position to end of line" function, which is an important factor in the efficient operation of many programs like the DynaStar high-speed screen editor. Other pluses include the multiple display windows and smooth scrolling.

The character drawing subroutines seem to be as fast as those in any of the popular implementations of Flex for the Color Computer, despite the penalties associated with allowing user-adjustable character sets. The speed of HiRes is equivalent to that of a standard terminal running at approximately 3200 Baud.

HiRes is very easy to run. You either load the character set and the module "HiRes," or insure that they are in your current execution directory. Then, you type "HiRes", and when the "OS9:" prompt appears next, it will be on your high-resolution screen.

HiRes is available from Frank Hogg Laboratory, 770 James Street, Syracuse, NY 13203.

WORD-PAK: A HARDWARE ALTERNATIVE TO HIRES SCREENS

There are several problems associated with the software approach to high resolution screens. The first concerns memory. It takes about 6,000 bytes of screen memory alone to generate a high resolution screen. After you add another two or three thou-

sand bytes to store the high resolution program, you'll find you have consumed quite a bit of memory. In fact, you'll have a lot less memory left for your own programs. Indeed, memory is a very precious commodity on an OS-9 Level I computer.

Speed is another consideration. The software approach uses a lot of your 6809's time — time that could be better used to run your own programs.

The hardware approach to high resolution screens is the answer. When you plug a cartridge containing a video generator chip and its own software into the Color Computer's expansion port you accomplish two things. You free up a lot of memory and you speed up your computer. You will also find that you are able to work with industry standard software designed to work on screens that sport 24 or 25 rows containing 80 columns.

Word-Pak, a video cartridge that plugs into your expansion port, gives you the complete ASCII character set, a programmable screen and a programmable cursor. The OS-9 device descriptor and device driver that works with it let you home the cursor, erase to the end of the line and erase to the end of the screen. They also let you position the cursor anywhere on the screen by sending a pair of X-Y coordinates.

Another model, the Word-Pak II, adds more advanced features. With this cartridge you get an improved character set that is very easy to read, smooth scrolling and a software video switch that lets you switch your monitor between the Color Computer's video output and its own. This means your programs can switch back and forth between graphics and text. Word-Pak and Word-Pak II were designed and built by PBJ, Inc.

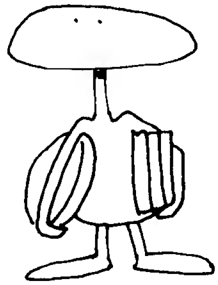
SDISK — A GOOD ALTERNATIVE TO CCDisk

OS-9 is a perfect example of the effectiveness of modular programming. You can add new hardware — additional disk drives, graphics display boards or mechanical plotters, for example — with ease. A package named SDISK from D. P. Johnson brings this ability to the Color Computer.

Suppose that you own 35-track, single-sided, single-density disk drives, and are struggling along doing the best you can. Then, your day comes and you win the Readers Digest Sweepstakes. Or, maybe you hit the daily double at the track. In any case, you'll probably want to move up to double-sided, double-density drives right away.

On a standard OS-9 system, you need only plug in the new drive and change one or two bytes in an OS-9 device descriptor. The change is so simple that you can make it quickly with the OS-9 debug utility.

Not so, on the Color Computer version of OS-9. Radio Shack hard-coded the description of their standard 35-track, single-sided, double-density drive into the device driver, CCDISK, instead of having the driver read the description from the device descriptor.



SDISK is a direct replacement for CCDisk. Once you have installed it, you can read and write 35, 40 or 80 track drives. Better yet, those drives can be single or double density, as well as single or double sided.

SDISK also lets you program the disk drive step rate. And, it gives you a way to read a disk written by a standard OS-9 system like the GIMIX. It also lets you write a disk that can be read by the larger systems.

ADDITIONAL UTILITIES FROM COMPUTERWARE

In addition to the Textools filter package discussed earlier in this chapter, Computerware of Encinitas, Calif., markets another utility package that gives you six programs designed to make OS-9 easier to run. It also includes a new device driver, named CCDisk, which replaces the original CCDISK module.

DirCopy is a very versatile backup program that lets you copy one disk to another, even if the formats are different. It lets you confirm the copying of each file, lets you copy sub-directories, pre-sorts the directory you are copying into alphabetical order and updates the file owner's number.

Patch lets you inspect and modify any file on a disk. It comes in handy when you need to change the value in a device descriptor in your OS9Boot file, but don't want to rebuild it. You can also use it to change the default data area requested by a program. It has a special Validate feature that lets you restore the header checksum and module CRC when you change a file. This is essential because if a modules CRC is incorrect, OS-9 refuses to load and execute it.

FileLook displays the size, type, revision number and name of all modules in a disk file. It's output looks a lot like an MDIR E, and the information it reports is very close to that provided by the Radio Shack Ident utility.

Compare lets you compare a module in memory to a module stored in a disk file. When there are differences, it will report the address of the differences.

Dmode lets you modify the device descriptors used to identify your disk drives. This makes it easy to access the additional features available on many drives. You can set the descriptors up for one- or two-sided drives, 6, 12, 20 or 30 millisecond stepping rates, up to 40 tracks per side. You must be using Computerware's new CCDisk to take advantages of these changes, however.

NewFmt is a replacement for the original Color Computer OS-9 format command that lets you create new single- or double-sided disks containing one to 40 tracks. This utility is interactive, and will let you determine the format of the disk before you execute the command.

CCDisk comes with a file that contains a script of Shell commands that automatically generate a new system disk for you. This new disk system disk will include Computerware's CCDisk in the OS9Boot file. Since Computerware gives you the command file, installation is a snap.

ANOTHER FILTER TIP

Every once in a while, I found myself getting behind in my writing. While I was preparing this book, I also found myself speaking at several RAINBOWfests around the country. Each time I got behind schedule, I would play catch up by writing on my Radio Shack Model 100 while riding in the carpool. What a tool!

However, when I uploaded the file from the Model 100, I found that it left the TAB character, 9 decimal, in the file. At other times, I've needed to remove the line feeds. For example, I often duplicate the output of a DIR command in this book by redirecting the output of the command to a file. Later, I merge the file into the text. Unfortunately, the DIR command sends out a line feed, 10 decimal or $0A hexadecimal, after the header.

These extra characters drive the cursor control logic in my screen editor bonkers. In fact, they may do the same to your screen editor. The solution is to use a TR or transliterate utility. One is available from the Users Group Software Committee, another is in the OS-9 File Handlers Toolbox from Microware described earlier in this chapter. Here's the command line I used.

OS9: list KISS.temp ! tr \9 ! tr \10 >KISS.December

EMULATING A TYPEWRITER

Here's another creative application for OS-9 that may save you some time. On my desk at work, I use a Wang PC. I didn't buy it. Frankly, I would rather use DynaStar or Stylo on OS-9 than the archaic, memory hungry, menu-driven-monster word processing software in the PC. However, the PC has one function I like a lot — it can emulate a typewriter.

Needless to say, you must select your way through two or three menus before you can use it, but the typewriter emulation on the PC really comes in handy for short notes and memos. There are many times when you just don't want to bother to go through three menus to create a new word processing document. I decided I would emulate this function on my GIMIX and Color Computer.

If you are running OS-9, you almost don't need to create it. The capability is built in. However, I want to take you through an experiment that will bring you greater understanding of a few OS-9 commands and a BASIC09 procedure that shows how you can use more than one technique to get a job done.

First, let's try to build a typewriter emulation with the copy command. It should work, shouldn't it? Try the command line below. Type the ESCAPE character — CLEAR BREAK — on Color Computer OS-9 when you get ready to quit.

OS9: copy /term /p

What happened? I'll bet it worked great on the first line you typed. You were probably even wearing a broad smile until you typed the second line. It was printed right on top of the first line, wasn't it?

This happens because the copy command does not use the built-in OS-9 line editing functions. It uses character input and output routines rather than line input and output. As any hacker will tell you, the OS-9 Copy utility uses the I$Read and I$Write system calls rather than the I$ReadLn and I$Writln calls.

Let's try something else. Type:

OS9: list /term >/p

It worked, didn't it? Congratulations! You now own a new pseudo typewriter — a typewriter that will let you edit or correct each line before you print it. Experiment a little and you will find that the CLEAR A, CLEAR X and other OS-9 special line edit keys all work while you are using this command line.

I was disappointed. I wanted to write a BASIC09 program to do the job. Actually, I wrote it anyway. It may just show you how certain high level language functions relate to functions at the Operating System level. Meet tw.

```
PROCEDURE tw

(* Emulate a typewriter *)

DIM printer:INTEGER
DIM in:STRING[80]

OPEN #printer,"/p":WRITE

LOOP
  ON ERROR GOTO 10
  INPUT "Enter: ",in
  WRITE #printer,in
ENDLOOP
```

```
10 CLOSE #printer
END
```

When you test "tw" you'll see that it works just like the second OS-9 command line above. But, since you now have your typewriter emulation written in a high level language you can add some fancy features of your own. Here's a HINT. Study the procedure CONVERT on Page 75 of The Official BASIC09 Tour Guide from Microware. Have fun!

OS-9 SOFTWARE VENDORS

Here is where you can order the tools discussed in this chapter.

OS-9 TEXTOOLS AND OTHER UTILITIES

Computerware
Box 668
Encinitas, CA 92024
Phone: 619-436-3512

WORK-PAK

PBJ, Inc.
P.O. Box 813
North Bergen, NJ 07047
Phone: 201-330-1898

UTILIX

Frank Hogg Laboratory
The Regency Tower, Suite 215
770 James Street
Syracuse, NY 13203
Phone: 315-474-7856

FILTER KIT AND HACKERS KITS

D. P. Johnson
7655 S.W. Cedarcrest Street
Portland, OR 97223
Phone: 503-244-8152

FILE HANDLER TOOL BOX

Microware Systems Corporation
5835 Grand Avenue
Des Moines, IA 50312
Phone: 515-244-1929

In this chapter you have been introduced to the concept of combining short programs — tools, so to speak — in OS-9 pipelines to do big jobs. Along the way, we've given you a description of most of the OS-9 toolkits available when this book went to press.

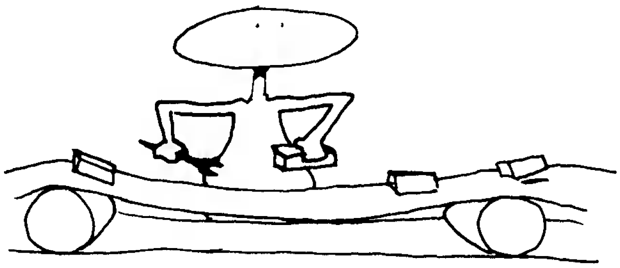
In Chapter 18 you'll meet the OS-9 assembler, ASM. And, we'll throw in a few programs for good measure.

CHAPTER 18

using the os-9 assembler

A complete tutorial on assembly language programming would take an entire book. In this chapter, we'll introduce you to ASM, the OS-9 assembler. In Part VI, we'll list several useful assembly language programs, including a complete device driver.

Here, we'll concentrate on a few of the programming techniques you'll need in all of your assembly language programs. By the end of this chapter you will learn how to:



Use the ASM command line
Create an OS-9 Module
Define character equates
Define strings
Print those strings
Get characters from the keyboard
Output a character
Output a Carriage Return and Linefeed
Output a decimal number
Use the OS-9 Get Status and Put Status calls
Run another OS-9 process from your program
Check for an End of File condition
Open a file for writing
Read a the decision tree from a menu

ASM — AN OVERVIEW

ASM is a 6809 assembler designed to work with OS-9. It lets you use special mnemonics to make OS-9 system calls. It automatically creates OS-9 module headers. And, it encourages you to develop position independent, reentrant code.

This table shows you what you need to do to develop an assembly language program.

1. Create a source code file using EDIT or your own editor.
2. Run ASM to translate the source code to 6809 object code.
3. If ASM reports errors, correct the source code with EDIT.
4. Use DEBUG to test the program.
5. If you find bugs, correct the source code with EDIT.
6. Document your work, so your program will be easy to use.

THE ASM COMMAND LINE

ASM, like other OS-9 commands, is run from the Shell. Here is the syntax of the ASM command line.

OS9: asm my_first_program [options] [#memsize] [>listing]

Everything in enclosed in brackets is optional. An actual command line looks like this.

OS9: asm my_first_program l s #16K >/p

This line will assemble the source code contained in a file named my_first_program in your current data directory. It will create a listing and symbol table and send them to your printer. Since you have used the OS-9 memory modifier option, ASM will use 16K of data memory.

Because of the length of the OS9Defs file that you must use in most of your assembly language programs, you will find that you need at least 12K to 16K to assemble a program. If you do not use the OS-9 memory modifier on the command line, ASM will usually report an error.

Since the OS-9 assembler is controlled by command line options, let's look at a few more.

OS9: asm #20K myfile o l >/p

The 'l' option on the command line causes the assembler to list the combined source and generate object code. The redirection operator, '>' causes the output to go to the printer. The 'o' option causes the assembler to send its output to a file named "myfile" in your current execution directory. This is usually /d0/CMDS. The printing can be turned on and off within the source code by using the opt statement, "opt l" or "opt -l".

You can redirect the object file by using the format shown in the command line below.

OS9: asm #20K myfile l o=/d1/mydirectory/myfile.obj

ASM will list the combined source and object code with the

proper tabbing, automatically. You must, however, make sure that each line that is supposed to start with a label starts in column one, and each line that starts with an operator starts in column two. If you want to make the entire line a remark, you may type an asterisk, ' * ', in column 1. Here's a useful 6809 subroutine shown before and after assembly.

SUBROUTINE SOURCE CODE BEFORE ASSEMBLY

```
* A routine to classify a character
* returns with carry set if character
* is not alphanumeric.

class cmpa #'z char is in a-reg
 bhi notasc
 cmpa #'a
 bhs ascii
 cmpa #'Z
 bhi notasc
 cmpa #'A
 bhs ascii
 cmpa #'9
 bhi notasc
 cmpa #'0
 bhs ascii
notasc orcc #1 set carry
 rts
ascii andcc #$FE clear carry
 rts
```

A LISTING OF THE SAME SUBROUTINE AFTER ASSEMBLY

```
Microware OS-9 Assembler 2.1

00001          * A routine to classify a character
00002          * returns with carry set if character
00003          * is not alphanumeric.
00004   0000 817A      class    cmpa  #'z      char is in a-reg
00005   0002 2214               bhi   notasc
00006   0004 8161               cmpa  #'a
00007   0006 2413               bhs   ascii
00008   0008 815A               cmpa  #'Z
00009   000A 220C               bhi   notasc
00010   000C 8141               cmpa  #'A
00011   000E 240B               bhs   ascii
00012   0010 8139               cmpa  #'9
00013   0012 2204               bhi   notasc
00014   0014 8130               cmpa  #'0
00015   0016 2403               bhs   ascii
00016   0018 1A01      notasc   orcc  #1       set carry
00017   001A 39                 rts
00018   001B 1CFE      ascii    andcc #$FE     clear carry
00019   001D 39                 rts
```

```
00000 error(s)
00000 warning(s)
$001E 00030 program bytes generated
$0000 00000 data bytes allocated
$0061 00097 bytes used for symbols
```

ASSEMBLY LANGUAGE TIPS

In the following tutorial we'll look at small pieces of code taken from a working program. We begin with the assumption that you have looked over the Radio Shack manuals and know a little about assembly language programming. We'll start at the beginning of a program.

CREATING A MODULE

```
          NAM SPELL
          IFP1
          USE /D0/DEFS/os9defs
          USE /D0/DEFS/li.equates
          ENDC

          TTL An OS-9 Utility to find misspelled words

prog MOD SPLEND,SPLNAM,PRGRM+OBJCT,REENT+1,SPELL,SPLMEM

     USE DPEQUATES

prvbuf rmb 32
lkbuf  rmb 32
       rmb 255 room for stack
SPLMEM equ .

SPLNAMFCS /SPELL/
COMF1  FCS '/d0/spell/common.dat'
mywf1  fcs '/d0/spell/MYWORDS.DAT'
DICTF1 FCS '/D0/spell/DICTIONARY.DAT'
shlstr fcs 'shell'
dirstr fcc /dir/
       fcb $0D

     USE SPELL.STRINGS
     EMOD

SPLEND EQU *
```

The first line of code tells ASM the name of the program. Several lines later, the pseudo operator, TTL, gives it some more information to print in the header of your listing.

Make special note of the lines that say, IFP1 ... ENDC. The IFP1 conditional tells ASM to use the lines between it and ENDC if

the assembler is on its first pass. That's why you never see the "USE /D0/DEFS/OS9DEFS" line in listings that are output from ASM. The program listing is generated during ASM's second pass.

The label PROG is used to tell ASM how to set up the module header in your program. If you read a lot of OS-9 programs, you'll notice that the MOD line in all programs is almost the same. Usually, only the name of the program you are reading has been changed.

SPLEND causes ASM to form a double byte equal to the length of the program. It will contain the offset from the beginning of the module to the label SPLEND. Remember, all OS-9 program modules must start at zero.

SPLNAM forms two bytes that hold the offset from the start of the module to the label, SPLNAM. The single bytes that follow tell OS-9 what type of program you are assembling.

At the end of the MOD line you'll see the labels SPELL and SPLMEM. SPELL causes ASM to form a double byte that contains the offset from the beginning of the module to the start of the executable code in the module, and SPLMEM tells OS-9 how much data memory area the program needs.

Notice that with OS-9 you always use two memory areas when you run a program. One area contains the program, the other contains the data used by the program. Nothing in the program area can change during execution. That is why all variables must be defined and stored in the data area. The line, "SPLMEM EQU ." tells ASM that this is the end of the data area.

Lines containing the USE pseudo operator tell ASM to insert the code contained in the file named. USE follows all standard OS-9 rules. If you give a filename only, ASM assumes it is stored in your current data directory. If you give a full pathlist, ASM will use that pathlist to find the file.

Since the DEFS files are stored in the DEFS directory on device /D0, you usually need to use a full path list to reach them. The other USE lines in the code above are filenames only; therefore, they are read from the current data directory. The source code this sample was taken from contains more than a dozen USE files.

Notice the line containing the EMOD operator. It tells OS-9 that this is the end of the program and automatically generates the modules CRC and inserts it at this point in the object code.

DEFINING CHARACTER EQUATES

```
*
* CHARACTER EQUATES
```

```
*
BELL EQU 7
LF EQU $A
CR EQU $D
SPACE EQU $20
NULL EQU 0
```

The code above is stored in a USE file called DPEQUATES. It holds the names and locations of all constants and variables used by the program. This means that you can edit a single constant once in this file, instead of changing it throughout the program.

After it reads the lines above, ASM substitutes the decimal number seven each time it sees the word BELL. Likewise, it will put the value 10 decimal or A hexadecimal in the object code every time it sees the word LF.

DEFINING STRINGS

```
RDICST FDB RDILEN
       FCB CR,LF,LF
       FCC /DynaSpell is looking for your words in its dictionary./
RDILEN EQU *-RDICST-2
```

This code segment shows you how to define a string using ASM. The label RDICST marks the beginning of a string definition.

When this string is printed you will see the words between the slashes on your terminal. The programming trick occurs at the label RDILEN. ASM computes the length of your string by subtracting the location of the beginning of the string and two additional bytes from its present location. It then stores this length at the label RDICST with the FDB pseudo operator.

PRINTING THOSE STRINGS

How do we print these strings? Follow this code:

```
pstr pshs a,y
  ldy ,x++
  lda opath
  os9 I$writ
  lbcs error
  puls a,y,pc
```

This subroutine does the job. We call it with the 6809's X-register pointing to the location of the length of the string, RDICST in this case. Before we begin, we push the value of the A-register and Y-register on to the stack. We're going to need them later.

Then, we load the Y-register with the length of the string. We do this by loading it with the value stored at an offset of zero from the X-register. Remember, we entered this routine with the X-

register pointing to — or containing — the location of RDICST, the string length.

Notice that when we loaded the length of the string in the Y-register we also incremented the X-register twice. This means that it is now pointing to the first character in our string. In our sample string, that character is a carriage return.

Finally, we load the OS-9 path number into the A-register and use an OS-9 system call, "OS9 I$WRIT. If there is an error, OS-9 will return with the carry set and we branch to a routine called error that takes care of the problem. Otherwise, we return by pulling the two registers saved earlier, and the program counter, off the stack. We just sent the string to the output path number contained in opath.

GETTING CHARACTERS FROM STANDARD INPUT PATH

How do you get a single character from the standard input path — keyboard? How do you print a single character, a carriage return and line feed, or even a decimal number? Study the routines that follow.

```
keyin pshs x,y,b,u
    bsr getchr
    pshs a
    puls x,y,b,a,u,pc

getchr pshs x,y
    leax chrbuf,u
    lda #0 standard input only
    ldy #1
    os9 I$read
    lbcs error
    lda chrbuf
    puls x,y,pc
```

To get a character, we call the routine keyin. It saves the 6809 registers on the stack, gets a character from the routine getchr, puts that character on the stack and returns by pulling all registers and the program counter. Our character is in the 6809's A-register when we return from keyin.

Getchr shows how to set up a routine to get a single character from the standard input path. Remember, most of the time you can equate standard input path with keyboard. When you use the OS9 I$READ call, the X-register must point to a buffer in memory where you are going to store the character. The A-register must contain the path number.

Remember, the standard input path is always zero. Notice, also, that I$READ leaves the character it returns stored at chrbuf. This means we must load it into the A-register before we return.

Check the description of the I$READ call in the blue Radio Shack OS-9 Technical Information Manual. You'll see that you can read any number of characters at a time. The number is put in the Y-register before the call. Getchr is a special case that only reads one character.

You can speed up your programs by reading or writing more than one character at a time.

SENDING CHARACTERS TO STANDARD OUTPUT

Now, let's take a look at a routine that puts out a single character.

```
* routine to output just one
* character to the standard output path

putchr pshs a,x,y
   leax chrbuf,u
   sta chrbuf
   lda opath
   ldy #1
   os9 i$writ
   lbcs error
   puls a,x,y,pc

pcrlf lda #cr
    lbsr putchr
    lda #lf
    lbra putchr hidden rts
```

These output routines are almost the direct opposite of the getchr routine. The only difference is that they use the OS9 I$WRIT call instead of I$READ.

Pcrlf gives you an easy way to output a carriage return and linefeed to the standard output path. The names of these routines should be familiar to most FLEX programmers. They are OS-9 routines that emulate equivalent FLEX subroutine calls.

PRINTING A DECIMAL NUMBER

```
* Routine to output a decimal
* number from the d-register

outdec pshs a,b,x
   leax dectab,pcr
   clr ,-s
   clr ,-s
dec3 clr ,s
dec2 inc ,s
   subd ,x
   bhs dec2
```

```
   addd ,x++
   pshs a,b
   lda 2,s
   deca
   tfr a,b
   orb 3,s
   stb 3,s
   beq dec4
   adda #'0
   bsr putchr
dec4 puls a,b
   tst 1,x
   bne dec3
   leas 2,s
   puls a,b,x,pc

dectab fdb 10000,1000,100,10,1,0
```

CHANGING A DEVICE DESCRIPTOR ON THE FLY

Here is a way to change the status of a device descriptor from within your assembly language program. For example, if you need to turn off the pause feature. Here's one way to do it.

```
* get status packet
* and set -pause and -lf

 clra
 clrb
 leax stapak,u
 os9 i$gstt
 lbcs error
 lda 7,x get pause condition
 sta pausav
 clr 7,x set no pause
 lda 5,x get lf condition
 sta lfsav
 clr 5,x set no auto line feeds
 clra path number
 clrb write status packet
 os9 i$sstt setstat call
 lbcs error
```

This routine uses the OS-9 get status call to retrieve the path descriptor that holds all device descriptor data for the current process. You must tell it where to save the information. We put it in a buffer named stapak,u. Once you have the information stored in a buffer you can modify it.

First, we retrieved the pause condition and saved it so that we could restore everything to the same condition when we leave our program. We know from the OS-9 Technical Information that the

pause attribute is stored at an offset of seven from the beginning of the path descriptor.

Then, we set the pause location to -pause, or zero, with the "clr 7,x" instruction, and did the same with the "lf" location. After storing the condition we wanted in our table, we copied that table back into the Path Descriptor table with the I$sstt, or set status call.

That takes care of the initial table change. However, when we leave our program we must put everything back the way it was. The routine below shows how we did it during a normal exit from the program. We used a similar routine in our error exit.

```
done lbsr clrscn clear screen before leaving
     leax stapak,u return pause
     lda lfsav and lf to prior
     sta 5,x condition before exiting
     lda pausav
     sta 7,x
     clra
     clrb
     os9 i$sstt do it!
     lbcs error
     clrb report no errors
     os9 f$exit
```

Have you ever wondered how to start another OS-9 process from within one of your own assembly language programs? Study this routine.

STARTING A NEW PROCESS

```
* now do dir
  leax shlstr,pcr "shell"
  ldy #4 size
  leau dirstr,pcr "dir"
  lda #1 object code
  clrb
  os9 f$fork
  lbcs error
  os9 f$wait
```

The code above sets up a call to a Shell that runs the OS-9 DIR utility. DIR must be either in memory or in your current execution directory when the code runs.

The instructions that load the registers show how you tell the OS-9 F$FORK system call what process you want to start. Leax shlstr,pcr points to a string in memory that holds the characters s-h-e-l-l. The eighth bit is set on the last 'l'. This tells OS-9 that it is at the end of a filename.

We point the U-register to the location of our parameter string — the name of the program we want our new Shell to execute. The string contains the letters, d-i-r, followed by a carriage return. You can see the actual strings in the first listing of this tutorial.

After we use the OS-9 F$FORK system call, we put our process to sleep and wait for the new Shell to die. To do this, we use the OS-9 F$WAIT system call.

Here's how it works. When the DIR command is finished, the Shell that ran it will die and send a signal to OS-9 to wake up the process that called it.

APPENDING DATA TO THE END OF A FILE

Here's a programming technique that lets you append data to the end of an existing file. The secret to its success is the OS-9 Get Status call described on Page 99 of the Radio Shack OS-9 Technical Information manual.

```
* First, load the 6809's A-register with path number
* and load B-register with SS.SIZ function code
* Then, use the OS-9 Get Status system call

    lda path
    ldb $02
    os9 I$getstt
    bcs error

* If there was no error, the most significant 16
* bits of the current file size will be in the
* X-register and the least significant 16 bits
* will be in the U-register.
```

To wrap up our assembly language tutorial we'll give you three more routines to study. You'll learn how to check for, and handle, an end-of-file condition, open a file, and make simple menu selections.

CHECKING FOR THE END OF A FILE

```
eofchk cmpb #e$eof end of file?
    lbne error no, go
    lda ipath yes, close read file
    os9 i$clos
    lbcs error
    lda opath standard output?
    cmpa #1
    beq eofc1 yes, go
    os9 i$clos no, close it
    lbcs error
    lda #1 and set up for
    sta opath standard output
```

```
eofc1 deca and standard input
   sta ipath
   orcc #1 set carry to indicate
   puls x,pc exit needed
```

OPENING A FILE FOR WRITE

```
* Open a file for write

writon clr reflag
  leax filnam,u open file
  lda #read.
  os9 i$open
  sta ipath
writ1 leax temstr,pcr now open "temp"
  ldd #write.*256+updat.+pread.+pwrit.
  os9 i$crea
  bcs wtemchk
  sta opath
```

READING DECISIONS FROM A MENU TREE

```
* A small segment from a menu selection
* decision tree

chkff cmpa #'f want formatted read?
   bne chku no, is character a "u"
   lbsr reasty yes, do formatted read
   lbra query and go back to main menu
chku cmpa #'u want to use another dictionary?
   bne chko no, see if want to quit
   lbsr use yes, go to it
   lbra query and return to main menu
chko cmpa #'o want to go back to operating system?
   bne chkfb no, want to save accepted words?
   lbra done back to OS-9 yes, go back
chkfb cmpa #'b build new dictionary list
   bne chkfw
   lbsr savwrd
   lbra query and the beat goes on
```

SAMPLE PROGRAMS

We conclude this chapter with several sample programs written in 6809 assembly language. They were contributed by Tim Harris, a computer science student at the University of Iowa at Ames.

Harris does all of his development on a Color Computer running under Radio Shack OS-9. These programs are written to be used like the filters described in Chapter 17. Study them! Then, type them in and run them. Experiment with them, and if you are

brave, modify them. In no time at all you'll be able to write your own OS-9 tools.

CRYPT

```
Microware OS-9 Assembler 2.1
crypt - OS-9 System Symbol Definitions

                * crypt utility : crypts files for user protection
                * 6809 Assembly Language
                * for Color Computer OS-9  v. 01.00.00
                * Contributed to the OS-9 Tour Guide by:
                * Tim Harris (c) 1984
                *
                * Uses std. input and output so it acts as a filter
                * Sample calls:
                *     crypt keyword <infile>codedfile
                *     crypt keyword <codefile      prints file to screen
                *     list infile ! crypt keyword >outfile
                *     crypt keyword <infile ! crypt keyword  will print out file
                *
                *
                                    nam   crypt
                                    use   /d0/DEFS/OS9Defs
                                    opt   l

                *
                * Data Area
       00D3               EOF       equ   211
       000F               MAXKEY    equ   15
     D 0000                         org   0
     D 0000               OUTCHAR   rmb   1
     D 0001               KEYLEN    rmb   1
     D 0002               CHAR      rmb   1
     D 0003               KEYBUF    rmb   MAXKEY
     D 0012                         rmb   200        stack area
     D 00DA               CRPMEM    equ   .
                * Program Area
       0000 87CD0061                mod   CRPEND,CRPNAM,PRGRM+OBJCT,REENT+1,CRPENT,
       000D 63727970      CRPNAM    fcs   "crypt"
       0012 5F            CRPENT    clrb             clear the counter
       0013 3143                    leay  KEYBUF,u   get the key value
       0015 A680          CRP10     lda   ,x+
       0017 810D                    cmpa  #$0D       are you done?
       0019 2709                    beq   CRP15      yes, go on with program
       001B 8120                    cmpa  #$20       maybe, check again?
       001D 2705                    beq   CRP15      yes, go on
       001F A7A0                    sta   ,y+        no, store the char
       0021 5C                      incb             incriment the counter
       0022 20F1                    bra   CRP10      go back for more
       0024 D701          CRP15     stb   KEYLEN     save the key length
       0026 D601          CRP20     ldb   KEYLEN     get key length
       0028 3143                    leay  KEYBUF,u   point to start of key
       002A 3424          CRP25     pshs  y,b
       002C 4F                      clra
       002D 108E0001                ldy   #$1
       0031 3042                    leax  CHAR,u
       0033 103F8B                  os9   I$Readln
       0036 251E                    bcs   CRP30
       0038 3524                    puls  b,y
       003A 9602                    lda   CHAR       get the char
       003C A8A0                    eora  ,y+        crypt it
       003E 9702                    sta   CHAR       store it for output
```

```
0040 3424                pshs  y,b
0042 8601                lda   #$1
0044 108E0001            ldy   #$1
0048 3042                leax  CHAR,u
004A 103F8C              os9   I$Writln
004D 2507                bcs   CRP30
004F 3524                puls  b,y
0051 5A                  decb             are you done?
0052 26D6                bne   CRP25      no, crypt more
0054 20D0                bra   CRP20      yes, get more of the file
0056 C1D3       CRP30    cmpb  #EOF       is it at EOF?
0058 2601                bne   CRP35      no, exit with error
005A 5F                  clrb             yes, clear error status
005B 103F06     CRP35    os9   F$Exit     exit the program
005E 9CC02A              emod
0061            CRPEND   equ   *
                         END

00000 error(s)
00000 warning(s)
$0061 00097 program bytes generated
$01B3 00435 data bytes allocated
$1024 04132 bytes used for symbols
```

DL

```
Microware OS-9 Assembler 2.1
dl - OS-9 System Symbol Definitions

       * dl utility: enhanced delete utility
       * modeled after OS-9/68K delete utility
       * CoCo OS-9  ver. RS01.00.00
       * Contributed to the OS-9 Tour Guide by:
       * (c) 1984 Tim Harris
       *     651 Pammel Court
       *     Ames, Iowa 50010
       *
       * Options:    -z  takes list of pathnames from stdin
       *             -?  generates help message
       *
       * Calls:      dl fname  deletes file named "fname"
       *             dl -?     gives help message
       *     with use of enhanced d (dir) utility you can:
       *              d ! dl -z   delete all entries in the dir
       *     d -w *.c ! dl -z   deletes all entries ending in ".c"
       *
                                use   /d0/DEFS/OS9Defs
                                opt   1

                                nam   dl
       * Data Area
  00D3                 EOF      equ   211
  003C                 NLEN     equ   60
D 0000                          org   0
D 0000                 FNAME    rmb   NLEN
D 003C                          rmb   200       stack size
D 0104                          rmb   200       param size
D 01CC                 DLMEM    equ   .
       * Module Macro
  0000 87CD00F6                 mod   DLEND,DLNAM,DLTYP,DLREV,DLENT,DLMEM
  000D 64EC            DLNAM    fcs   "dl"
  0011                 DLTYP    set   PRGRM+OBJCT
```

```
0081                DLREV     set    REENT+1
          * Program Area
000F A684           DLENT     lda    0,x          get first param character
0011 810D                     cmpa   #$0D         is it c.r.?
0013 2602                     bne    DL05         no, branch around
0015 8D23                     bsr    help         yes, send help message, improp
0017 812D           DL05      cmpa   #'-          is it a minus sign?
0019 2610                     bne    DL20         no, use regular param delete r
001B A601                     lda    1,x          yes, get next char
001D 817A                     cmpa   #'z          is it a 'z'?
001F 2602                     bne    DL10         no, check for other option
0021 8D29                     bsr    zoption      yes, use z routine
0023 815A           DL10      cmpa   #'Z          is it a 'Z'?
0025 2602                     bne    DL15         no, check for other option
0027 8D23                     bsr    zoption      yes, use z routine
0029 8D0F           DL15      bsr    help         go to help if ? or illegal opt
002B 103F87         DL20      os9    I$Delete     delete the file
002E 2507                     bcs    DL25         branch on error
0030 A684                     lda    0,x
0032 810D                     cmpa   #$0D         is it end of param list?
0034 26F5                     bne    DL20         no, delete more files
0036 5F                       clrb                clear error status
0037 103F06         DL25      os9    F$Exit       end program
          * Subroutines
003A 8601           help      lda    #$1          set std. out
003C 308D0028                 leax   HMSG,pcr     get message
0040 108E008B                 ldy    #HMLEN       get length
0044 103F8A                   os9    I$Write      write it out
0047 2501                     bcs    h05          branch on error
0049 5F                       clrb                clear error status
004A 20EB           h05       bra    DL25         exit program
          *
004C 4F             zoption   clra                set std. input
004D 30C4                     leax   FNAME,u      point to buffer
004F 108E003C                 ldy    #NLEN        get max length
0053 103F8B                   os9    I$Readln     read it in
0056 2509                     bcs    z05          branch on error
0058 30C4                     leax   FNAME,u      point to name again
005A 103F87                   os9    I$Delete     delete the file
005D 25D8                     bcs    DL25         exit on error
005F 20EB                     bra    zoption      get more if no error
0061 C1D3           z05       cmpb   #EOF         is it EOF?
0063 2601                     bne    z10          no, error out
0065 5F                       clrb                yes, clear status
0066 20CF           z10       bra    DL25         exit program
          *
          * Help message
0068 0D0A           HMSG      fcb    $0d,$0a
006A 20646C3A                 fcc    / dl: deletion utility/
007F 0D0A                     fcb    $0d,$0a
0081 206F7074                 fcc    / options: -z get paths from list on stdi
00AA 0D0A                     fcb    $0d,$0a
00AC 20202020                 fcc    /          -? help message/
00C5 0D0A                     fcb    $0d,$0a
00C7 2043616C                 fcc    / Call:   dl <options> || <fname>...<fnam
00F1 0D0A                     fcb    $0d,$0a
008B                HMLEN     equ    *-HMSG
00F3 173DCE                   emod
00F6                DLEND     equ    *
                              END

00000 error(s)
```

```
00000 warning(s)
$00F6 00246 program bytes generated
$02A5 00677 data bytes allocated
$106F 04207 bytes used for symbols
```

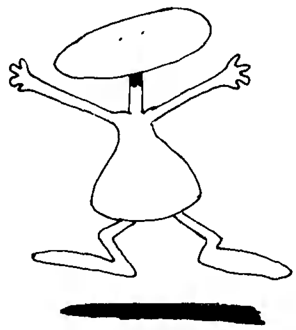
SUMMARY

In this chapter you have been introduced to ASM, the OS-9 assembler. We have also presented a few assembly language techniques that you can use every day.

Study these techniques for awhile. Then, join us in Chapter 19 for a look at BASIC09, C and Pascal.

CHAPTER 19

high level language



From the beginning, we have tried to impress upon you that OS-9 is an environment, a home for programming tools of all kinds. OS-9 tools range from the filters we introduced in Chapter 17 to the assembler we met in the last chapter. They extend to the high level languages we study here.

OS-9 is also a home for the many application programs we use in our businesses and homes daily. These applications range from database managers to spread sheets to word processors, with a random sampling of games and other diversions thrown in for variety.

These applications all have one thing in common. They were developed using the assembler, or one of the high level languages introduced in this chapter. Once you begin to understand these languages and how they live within OS-9, you will be able to write your own applications. When you finish this chapter you will have met three high-level languages and studied several programs in each. You'll meet:

BASIC09 C Pascal

BASIC09 — THE REASON BEHIND OS-9

Yes, OS-9 was developed for a reason. Motorola and Microware needed a sophisticated operating system to provide a suitable environment for a state of the art language — BASIC09.

Why has BASIC09 been heralded as a state of the art language? First, it gives you a tool to write structured, readable

programs in a friendly environment. And second, because it contains a built-in editor and debugging aids, it is also highly interactive.

The structure of a BASIC09 program is made possible by the control statements. All the common loop constructs are available: IF/THEN/ELSE, WHILE/ENDWHILE, REPEAT/UNTIL, LOOP/ENDLOOP, AND EXITIF/ENDEXIT. With control statements like these, you can throw away those confusing line numbers forever. Yet, if you really need them, you can use them.

And, just as OS-9 uses small modules of code to build a complete operating system, BASIC09 lets you write small pieces of code that you can use over and over again in many different programs. BASIC09 calls these small pieces procedures.

BASIC09 lets you define your own data types by combining its primitive data types. This encourages you to divide your programs into small procedures because it makes it very easy to pass parameters between procedures.

CHECK OUT THIS BENCHMARK

And, BASIC09 is fast. To find out how fast, I tried one of the old BASIC benchmark programs with BASIC09.

I picked the infamous Benchmark Program Number Seven, written by Tom Rugg and Phil Feldman and first published in Kilobaud back in June of 1977. This is the program that took 204.5 seconds to execute on Southwest Technical Products 8K BASIC. The fastest BASIC, running at two megahertz on an OSI Challenger, executed this benchmark in 21.6 seconds.

Here's the BASIC09 procedure.

```
PROCEDURE  Benchmark
0000       DIM a,l,k:INTEGER
000F       DIM m(5):INTEGER
001B       PRINT "start"
0024       SHELL "date,t"
002E       k=0
0035 10    k=k+1
0043       a=k/2*3+4-5
0057       GOSUB 20
005B       FOR l=1 TO 5
006B         m(l)=a
0077       NEXT l
0082       IF k<1000 THEN 10
0092       SHELL "date,t"
009C       END
009E 20    RETURN
```

BASIC09, running at two megahertz on a GIMIX microcompu-

ter, finished the program in four seconds. It only took eight seconds using BASIC09 on the Color Computer. By comparison XBASIC, from TSC, long billed as the fastest BASIC interpreter on an eight-bit microprocessor, executed the benchmark in 24 seconds on the GIMIX.

Since I defined all the variables in the BASIC09 program as INTEGERs, I decided to play fair and rewrote the XBASIC program to use integers. After I did this, it took 12 seconds to run. BASIC09 still ran three times faster.

I decided to make one more comparison and rewrote the BASIC09 program with REAL variables. It executed in nine seconds; still more than twice as fast as in XBASIC using real variables as loop counters.

BASIC and BASIC09 — COMPARED

If you have ever taken a writing course, you'll remember the instructor telling you to "show" and not "tell." For this reason, we're finishing our tour of BASIC09 with 10 short feature programs.

Actually, you'll find 20 program listings because we have taken 10 relatively simple BASIC programs and written them in two different ways. The first listing uses standard BASIC — the second, BASIC09.

The structured, highly readable, self-documenting style of BASIC09 programs should speak for itself. In this Chapter you'll find the listings named below. The BASIC09 listings use the same name with an "09" appended. The last three programs show you how you can write filter programs in BASIC09. NewStrip, SplitWords and New_Hex_Dump can all three be used in an OS-9 pipeline, like the other filters demonstrated in Chapter 17.

NumberGuess	CoinFlip	PowersOfTwo	PrimeNumbers
DecimalToBinary	Acey_Deucy	Quiz	Temperature
Regular_Deposits	ESP	NewStrip	SplitWords
New_Hex_Dump			

These programs will give you a chance to gain some experience with BASIC09. They were printed directly from a BASIC09 listing, so you shouldn't have any problems running them.

We kept our samples short so they would be easy for you to type in and run. They are all simple programs that only begin to use the power of this programming language. Before you try to write lengthy programs of your own in BASIC09, try customizing the programs listed here. It will give you a good feel for the language. But most of all — Enjoy!

NumberGuess

This program uses the computer to generate random numbers. Your job is to guess the number. After you have made the correct guess, the computer will tell you how many attempts you needed to arrive at the answer.

NumberGuess

```
PROCEDURE numberguess
 0000 100  REM Number Guessing Game
 001A 110  REM To stop type "Control C"
 0038 120  PRINT "I will think of a number "
 0058 130  PRINT "between 1 and 100 -- "
 0074 140  PRINT "You try to guess it! "
 0090 150  n=0
 009B 160  x=INT(RND(0)*99+1)
 00B0 170  PRINT
 00B5 180  PRINT "What's your guess";
 00CE 190  INPUT g
 00D6 200  n=n+1
 00E5 210  PRINT
 00EA 220  IF g=x THEN 280
 00FD 230  IF g>x THEN 260
 0110 240  PRINT "Too small, try again";
 012C 250  GOTO 190
 0133 260  PRINT "Too large, try again";
 014F 270  GOTO 190
 0156 280  PRINT "You got it in "; n; " tries."
 0179 290  IF n>6 THEN 310
 018C 300  PRINT "Very good!"
 019D 310  PRINT
 01A2 320  PRINT
 01A7 330  GOTO 120
 01AE 340  END
```

NumberGuess09

```
PROCEDURE numberguess09
 0000
 0001       REM Number Guessing Game
 0018
 0019       DIM numberofguesses,randomnumber,yourguess:INTEGER
 0028
 0029       PRINT
 002B       PRINT "I will think of a number between 1 and 100."
 005A       PRINT "Let's see if you can guess what it is."
 0084
 0085       numberofguesses:=0
 008C       randomnumber:=INT(RND(1)*99+1)
 009F
 00A0       LOOP
 00A2
 00A3       EXITIF randomnumber=yourguess THEN
 00B0         PRINT
 00B2       ENDEXIT
 00B6
 00B7         numberofguesses:=numberofguesses+1
 00C2
 00C3         PRINT
 00C5
 00C6         INPUT "What's your guess? ",yourguess
 00E1
 00E2         IF yourguess>randomnumber THEN
 00EF           PRINT "Too high!  Try Again!"
```

```
0108         ELSE
010C           PRINT "Too low!  Try Again!"
0124         ENDIF
0126       ENDLOOP
012A
012B       PRINT
012D
012E       IF numberofguesses<6 THEN
013A         PRINT "Very Good!"
0148       ENDIF
014A
014B       PRINT
014D       PRINT "You got it in "; numberofguesses; " tries."
016D       PRINT "Thank you for playing!"
0187       PRINT
0189
018A       END
```

CoinFlip

Here's another program that uses the computers ability to generate random numbers. We let it flip a coin 50 times so that we may study the long-term average. You'll find that you wind up with heads about 50 percent of the time. Now there's an even bet.

```
PROCEDURE Coinflip
 0000 100  REM Determine number of heads
 001F 110  REM or tails in 50 flips of a coin
 0043 120  y=1
 004E 130  c=0
 0059 140  x=1
 0064 150  f=INT(RND(0)*2)
 0075 160  IF f=1 THEN 190
 0088 170  PRINT "T";
 0091 180  GOTO 210
 0098 190  c=c+1
 00A7 200  PRINT "H";
 00B0 210  x=x+1
 00BF 220  IF x<51 THEN 150
 00D2 230  PRINT
 00D7 240  PRINT c; " Heads out of 50 flips."
 00F9 250  PRINT
 00FE 260  PRINT
 0103 270  y=y+1
 0112 280  IF y<11 THEN 130
 0125 290  END
```

CoinFlip09

```
PROCEDURE Coinflip09
 0000
 0001      (* Count number of heads and tails *)
 0026      (* during 50 flips of a coin *)
 0045
 0046      DIM flip,heads,coins,groups:INTEGER
 0059
 005A      groups:=1
 0061
 0062      WHILE groups<11 DO
 006E
 006F        heads:=0
```

```
0076        coins:=1
007D
007E        WHILE coins<51 DO
008A
008B          flip:=INT(RND(0)*2)
009A
009B          IF flip=1 THEN
00A7            PRINT "H";
00AD            heads:=heads+1
00B8          ELSE
00BC            PRINT "T";
00C2          ENDIF
00C4
00C5          coins:=coins+1
00D0
00D1        ENDWHILE
00D5
00D6        PRINT
00D8        PRINT heads; " Heads our of 50 flips."
00F7        PRINT
00F9
00FA        groups:=groups+1
0105
0106      ENDWHILE
010A
```

ESP

This is another variation on a number-guessing program. You're trying to guess whether the computer will get a "head" or a "tail" when it flips a coin. As you play, the computer keeps a tab on your answers — both right and wrong.

```
PROCEDURE esp
 0000 100  REM esp tester
 0010 110  REM Type E to stop
 0024 120  h=1
 002F 130  w=0
 003A 140  t=0
 0045 150  c=0
 0050 160  e=10
 005B 170  f=INT(RND(0)*2)
 006C 180  IF f=0 THEN a$="H" \ ENDIF
 0085 190  IF f=1 THEN a$="T" \ ENDIF
 009E 200  PRINT "H or T";
 00AC 210  INPUT x$
 00B4 220  PRINT
 00B9 230  IF x$=a$ THEN 280
 00CC 240  IF x$="E" THEN 330
 00DF 250  w=w+1
 00EE 260  PRINT "Wrong!"
 00FB 270  GOTO 300
 0102 280  c=c+1
 0111 290  PRINT "Right!"
 011E 300  PRINT "W="; w; " R="; c
 0135 310  PRINT
 013A 320  GOTO 170
 0141 330  PRINT "Bye"
 014B 340  END
```

ESP09

```
PROCEDURE esp09
 0000
 0001       (* Outguess the computer *)
 001C
 001D       DIM correct,wrong,flip:INTEGER
 002C       DIM computer,guess:STRING[1]
 003C
 003D       correct:=0
 0044       wrong:=0
 004B
 004C       REPEAT
 004E
 004F         flip:=INT(RND(0)*2)
 005E
 005F         IF flip=0 THEN
 006B           computer:="H"
 0073         ELSE
 0077           computer:="T"
 007F         ENDIF
 0081
 0082         INPUT "Your guess -- (H)eads or (T)ails? ",guess
 00AC
 00AD         IF computer=guess THEN
 00BA           correct:=correct+1
 00C5           PRINT "Right!"
 00CF         ELSE wrong:=wrong+1
 00DD           PRINT "Wrong!"
 00E7         ENDIF
 00E9
 00EA         PRINT
 00EC         PRINT "Number wrong = "; wrong
 0103         PRINT "Number right = "; correct
 011A         PRINT
 011C
 011D       UNTIL guess="E" OR guess="e"
 0131
 0132       PRINT "Goodbye"
 013D       PRINT
 013F
 0140       END
 0142
```

PowersOfTwo

Here's a program that generates powers of two until it reaches the mathematical limit of BASIC09. It shows how BASIC09 automatically prints numbers in scientific notation when they become longer than six digits.

```
PROCEDURE powersoftwo
 0000 100  REM Print powers of two
 0019 110  PRINT
 001E 120  PRINT "Powers of two"
 0032 130  PRINT
 0037 140  PRINT "Power value"
 0049 150  x=0
 0054 160  y=1
 005F 170  PRINT x,y
 006B 180  y=y*2
 007A 190  x=x+1
 0089 200  IF x=125 THEN 220
 009C 210  GOTO 170
 00A3 220  END
```

```
PROCEDURE powersoftwo09
 0000
 0001       (* Generate powers of two *)
 001D       (* till the computer throws in the towell *)
 0049
 004A       DIM number:INTEGER
 0051       DIM powersquared:REAL
 0058
 0059       PRINT
 005B       PRINT "Powers of two"
 006C       PRINT
 006E
 006F       number:=0
 0076       powersquared:=1
 007E
 007F       WHILE number<125 DO
 008B
 008C         PRINT number,powersquared
 0095         powersquared:=powersquared*2
 00A1         number:=number+1
 00AC
 00AD       ENDWHILE
 00B1
 00B2       END
```

DecimalToBinary

Now you can generate binary numbers — ones and zeros —with your computer. This program takes a decimal number and prints it as a binary number. If you experiment, it will help you understand what's going on inside your computer.

```
PROCEDURE decimaltobinary
 0000 100  REM Decimal to binary converter
 0021 110  PRINT
 0026 120  PRINT "Decimal to binary converter"
 0048 130  PRINT
 004D 140  PRINT
 0052 150  PRINT
 0057 160  INPUT x
 005F 170  IF x<0 THEN 320
 0072 180  IF x>32767 THEN 320
 0086 190  PRINT
 008B 200  PRINT "X=";
 0095 210  y=16384
 00A1 220  a=INT(x/y)
 00B1 230  IF a=0 THEN 270
 00C4 240  PRINT "1";
 00CD 250  x=x-y
 00DC 260  GOTO 280
 00E3 270  PRINT "0";
 00EC 280  y=y/2
 00FB 290  IF INT(y)=0 THEN 310
 010F 300  GOTO 220
 0116 310  GOTO 140
 011D 320  END
```

Decimal To Binary09

```
PROCEDURE decimaltobinary09
 0000
 0001      (* Convert decimal numbers to binary *)
 0028
 0029      DIM number,temp,halfnumber:INTEGER
 0038
 0039      PRINT
 003B      PRINT "Decimal to Binary Converter"
 005A      PRINT
 005C
 005D      LOOP
 005F        PRINT
 0061        INPUT "What number would you like to convert? ",number
 0090
 0091      EXITIF number<0 OR number>32767 THEN
 00A5        PRINT "Negative numbers not allowed."
 00C6      ENDEXIT
 00CA
 00CB        PRINT
 00CD        PRINT "Number = ";
 00DB        halfnumber:=16384
 00E3        REPEAT
 00E5          temp:=INT(number/halfnumber)
 00F3          IF temp<>0 THEN
 00FF            PRINT "1";
 0105            number:=number-halfnumber
 0111          ELSE
 0115            PRINT "0";
 011B          ENDIF
 011D          halfnumber:=halfnumber/2
 0128        UNTIL INT(halfnumber)=0
 0135      ENDLOOP
 0139
```

PrimeNumbers

This program generates prime numbers until you tell it to stop.

```
PROCEDURE primenumbers
 0000 100  REM Prime Number generator
 001C 110  PRINT "Prime Number Generator"
 0039 120  y=2
 0044 130  a=1
 004F 140  GOTO 210
 0056 150  x=1
 0061 160  x=x+1
 0070 170  z=INT(y/x)
 0080 180  IF INT(z*x)=y THEN 230
 0098 190  IF x*x>y THEN 210
 00AF 200  GOTO 160
 00B6 210  PRINT a,y
 00C2 220  a=a+1
 00D1 230  y=y+1
 00E0 240  GOTO 150
 00E7 250  END
```

PrimeNumbers09

```
PROCEDURE primenumbers09
 0000
 0001      (* Generate and prime numbers *)
 0021
```

```
0022      DIM count,prime,temp,holder:INTEGER
0035
0036      count:=1
003D      prime:=2
0044
0045      PRINT count,prime
004E
004F      count:=count+1
005A      prime:=prime+1
0065
0066      LOOP
0068
0069        temp:=1
0070 100    temp:=temp+1
007E        holder:=INT(prime/temp)
008C
008D        IF INT(holder*temp)=prime THEN
00A0        ELSE
00A4
00A5          IF temp*temp>prime THEN
00B6            PRINT count,prime
00BF            count:=count+1
00CA          ELSE
00CE            GOTO 100
00D2          ENDIF
00D4
00D5        ENDIF
00D7
00D8        prime:=prime+1
00E3
00E4      ENDLOOP
00E8
00E9      END
```

Acey_Deucy

Games are a good diversion, and this one should be fun to play. It's a little longer than the others, so make sure you save it on disk before you move on. Typing is no fun.

```
PROCEDURE Acey_Deucy
 0000 100  REM Game of Acey-Deucy
 0018 110  PRINT "Acey-Deucy"
 0029 120  PRINT "You will get 25 hands."
 0046 130  PRINT
 004B 140  PRINT "Will the card you draw"
 0068 150  PRINT "be between my two?"
 0081 160  PRINT
 0086 170  h=1
 0091 180  PRINT
 0096 190  t=100
 00A1 200  PRINT "You have $"; t
 00B6 210  x=INT(7*RND(0)+6)
 00CB 220  IF x>12 THEN 210
 00DE 230  y=INT(x*RND(0)+1)
 00F3 240  IF y>=x THEN 230
 0106 250  IF y=1 THEN y=2 \ ENDIF
 011F 260  a=x
 012A 270  GOSUB 600
 0131 280  a=y
 013C 290  GOSUB 600
 0143 300  PRINT
```

```
0148 310  PRINT "Your bet";
0158 320  INPUT b
0160 330  IF b<=t THEN 360
0173 340  PRINT "You don't have that much!"
0193 350  GOTO 310
019A 360  z=INT(13*RND(0)+2)
01AF 370  IF z>14 THEN 360
01C2 380  a=z
01CD 390  GOSUB 600
01D4 400  PRINT
01D9 410  IF z<=y THEN 480
01EC 420  IF z>=x THEN 480
01FF 430  PRINT "You win!"
020E 440  PRINT
0213 450  PRINT
0218 460  t=b+t
0227 470  GOTO 530
022E 480  PRINT "You lose!"
023E 490  PRINT
0243 500  PRINT
0248 510  t=t-b
0257 520  IF t<=0 THEN 560
026A 530  h=h+1
0279 540  IF h>25 THEN 580
028C 550  GOTO 200
0293 560  PRINT "Your out!"
02A3 570  STOP
02A8 580  PRINT "That's 25 hands!"
02BF 590  STOP
02C4 600  IF a<11 THEN 630
02D7 610  IF a>14 THEN PRINT "ERROR" \ STOP  \ ENDIF
02F3 620  ON a-10 GOTO 650,670,690,710
0312 630  PRINT a; " ";
031F 640  RETURN
0324 650  PRINT "Jack ";
0331 660  RETURN
0336 670  PRINT "Queen ";
0344 680  RETURN
0349 690  PRINT "King ";
0356 700  RETURN
035B 710  PRINT "Ace ";
0367 720  RETURN
036C 730  END
```

AceyDeucy09

```
PROCEDURE AceyDeucy09
 0000
 0001      (* The Game of Acey - Deucy *)
 001F
 0020      DIM firstcard,secondcard,deal:INTEGER
 002F      DIM bet,hand,ante,yourdraw:INTEGER
 0042
 0043      PRINT "Acey - Deucy"
 0053      PRINT "You will get 25 hands."
 006D      PRINT
 006F      PRINT "Will the card you draw"
 0089      PRINT "be between my two?"
 009F      PRINT
 00A1
 00A2      hand:=1
 00A9      ante:=100
 00B0
 00B1      WHILE hand<=25 DO
 00BD
```

```
00BE        PRINT "You have $"; ante
00D0
00D1        REPEAT
00D3          firstcard:=INT(7*RND(0)+6)
00E6        UNTIL firstcard<13
00F1
00F2        REPEAT
00F4          secondcard:=INT(firstcard*RND(0)+1)
0108        UNTIL secondcard<firstcard
0114
0115        IF secondcard=1 THEN
0121          secondcard:=2
0128        ENDIF
012A
012B        deal:=firstcard
0133        GOSUB 200
0137
0138        deal:=secondcard
0140        GOSUB 200
0144
0145 100    PRINT
014A        INPUT "Your bet? ",bet
015C
015D        IF bet>ante THEN
016A          PRINT "You don't have that much!"
0187          GOTO 100
018B        ENDIF
018D
018E        REPEAT
0190          yourdraw:=INT(13*RND(0)+2)
01A3        UNTIL yourdraw<15
01AE
01AF        deal:=yourdraw
01B7        GOSUB 200
01BB
01BC        PRINT
01BE        IF yourdraw<=secondcard OR yourdraw>=firstcard THEN
01D3          PRINT "You lose!"
01E0          PRINT  \ PRINT
01E4          ante:=ante-bet
01F0        ELSE
01F4          PRINT "You Win!"
0200          PRINT  \ PRINT
0204          ante:=ante+bet
0210        ENDIF
0212
0213        IF ante<=0 THEN
021F          PRINT "You're out!"
022E          END
0230        ENDIF
0232
0233        hand:=hand+1
023E
023F      ENDWHILE
0243
0244      PRINT "That's 25 hands!"
0258      END
025A
025B 200  IF deal<11 THEN
026A        PRINT deal; "    ";
0276        RETURN
0278      ENDIF
027A
027B      IF deal>14 THEN
```

```
0287         PRINT "Error"
0290         END
0292       ENDIF
0294
0295       ON deal-10 GOTO 300,400,500,600
02AF
02B0 300   PRINT "Jack ";
02BD       RETURN
02BF
02C0 400   PRINT "Queen ";
02CE       RETURN
02D0
02D1 500   PRINT "King ";
02DE       RETURN
02E0
02E1 600   PRINT "Ace ";
02ED       RETURN
02EF
02F0       END
```

Quiz

Here's a program you can use to teach your youngsters the multiplication tables. It shows how computers can be used as teaching aids.

```
PROCEDURE quiz
 0000 100  REM Multiplication Quiz
 0019 110  PRINT "Multiplication Quiz"
 0033 120  n=0
 003E 130  c=0
 0049 140  i=0
 0054 150  x=INT(RND(0)*13+1)
 0069 160  y=INT(RND(0)*13+1)
 007E 170  z=x*y
 008D 180  PRINT
 0092 190  PRINT x; " times "; y; "=";
 00AD 200  INPUT w
 00B5 210  PRINT
 00BA 220  IF w=z THEN 270
 00CD 230  PRINT "Whoops!"
 00DB 240  PRINT "The answer is "; z
 00F4 250  i=i+1
 0103 260  GOTO 290
 010A 270  PRINT "You are right!"
 011F 280  c=c+1
 012E 290  PRINT c; " are right!"
 0144 300  PRINT i; " are wrong!"
 015A 310  n=n+1
 0169 320  IF n<=9 THEN 150
 017C 330  IF c>=6 THEN 370
 018F 340  PRINT "You flunked the quiz."
 01AB 350  PRINT "Better practice!"
 01C2 360  GOTO 120
 01C9 370  IF c>=9 THEN 400
 01DC 380  PRINT "You did O.K."
 01EF 390  GOTO 410
 01F6 400  PRINT "Nice Job!"
 0206 410  PRINT "Try again?"
 0217 420  INPUT T$
 021F 430  IF T$="y" THEN 120
 0232 440  IF T$="Y" THEN 120
 0245 450  END
```

```
PROCEDURE quiz09
 0000
 0001      (* A Multiplicaton Drill *)
 001C
 001D      DIM counter,correct,wrong,guess:INTEGER
 0030      DIM firstnumber,secondnumber,computersanswer:INTEGER
 003F      DIM query:STRING[1]
 004B
 004C      counter:=0
 0053      correct:=0
 005A      wrong:=0
 0061
 0062      WHILE counter<=9 DO
 006E
 006F        firstnumber:=INT(RND(0)*13+1)
 0082        secondnumber:=INT(RND(0)*13+1)
 0095        computersanswer:=firstnumber*secondnumber
 00A1
 00A2        PRINT
 00A4        PRINT firstnumber; " times "; secondnumber; " = ";
 00BE        INPUT guess
 00C3        PRINT
 00C5
 00C6        IF guess=computersanswer THEN
 00D3          PRINT "You are right!"
 00E5          correct:=correct+1
 00F0        ELSE
 00F4          PRINT "Whoops!"
 00FF          PRINT "The correct answer is "; computersanswer
 011D          wrong:=wrong+1
 0128        ENDIF
 012A
 012B        PRINT
 012D        PRINT correct; " are right."
 0140        PRINT wrong; " are wrong."
 0153        PRINT
 0155
 0156        counter:=counter+1
 0161
 0162      ENDWHILE
 0166
 0167      IF correct>=9 THEN
 0173        PRINT "Nice Job!"
 0180      ELSE
 0184        IF correct>=6 THEN
 0190          PRINT "You did O. K."
 01A1        ELSE
 01A5          PRINT "You flunked the quiz."
 01BE          PRINT "Better Practice!"
 01D2        ENDIF
 01D4      ENDIF
 01D6
 01D7      PRINT
 01D9      INPUT "Try Again? ",query
 01EC
 01ED      IF query="Y" OR query="y" THEN
 0202        RUN quiz09
 0206      ENDIF
 0208
 0209      END
```

Temperature

Now you can have BASIC09 help you. If you know the temperature in Fahrenheit, you can compute it in Centigrade. The reverse is also true. It's much faster than the bank thermometer.

```
PROCEDURE temperature
 0000 100  REM Convert temperatures
 001A 110  PRINT "This program converts temperatures -- "
 0047 120  PRINT "Fahrenheit to Centigrade and Vice-Versa"
 0075 130  PRINT
 007A 140  PRINT "Type the temperature you want to convert"
 00A9 150  PRINT "followed by comma and a"
 00C7 160  PRINT
 00CC 170  PRINT " 0 to get Fahrenheit"
 00E7 180  PRINT " 1 to get Centigrade"
 0102 190  PRINT
 0107 200  PRINT "Type a 'Control Q' to quit."
 0129 210  PRINT
 012E 220  c=0
 0139 230  f=1
 0144 240  INPUT x,y
 0150 250  IF y>1 THEN 240
 0163 260  IF y=1 THEN 310
 0176 270  a=9*x/5+32
 018D 280  PRINT "    ="; a; " Fahrenheit"
 01AA 290  PRINT
 01AF 300  GOTO 240
 01B6 310  a=5*(x-32)/9
 01CD 320  PRINT "    ="; a; " Centigrade"
 01EA 330  PRINT
 01EF 340  GOTO 240
 01F6 350  END
```

Temperature09

```
PROCEDURE temperature09
 0000
 0001       (* This program converts temperatures *)
 0029
 002A       DIM temperature:REAL
 0031       DIM preferred:STRING[1]
 003D
 003E       PRINT "This program converts temperatures -- "
 0068       PRINT "Fahrenheit to Centigrade and Vice-Versa."
 0094       PRINT
 0096
 0097       LOOP
 0099         PRINT
 009B         PRINT "Type a 'Control Q' to quit."
 00BA         INPUT "What is the temperature you would like to convert? "
              ,temperature
 00F5 10      INPUT "To (F)ahrenheit or (C)entigrade? ",preferred
 0121
 0122         PRINT
 0124
 0125         IF preferred="C" OR preferred="c" THEN
 013A           temperature:=5*(temperature-32)/9
 014E           PRINT "    ="; temperature; " Centigrade."
 0169         ELSE
 016D           IF preferred="F" OR preferred="f" THEN
 0182             temperature:=9*temperature/5+32
 0196             PRINT "    ="; temperature; " Fahrenheit."
 01B1           ELSE
 01B5             GOTO 10
```

```
01B9          ENDIF
01BB        ENDIF
01BD      ENDLOOP
01C1      END
01C3
```

Regular_Deposits

If you like to daydream about being rich, here's a program to get you started. Tell the computer how much money you would like to deposit, give it the present interest rate and tell it how long you would like to save. It will tell you what your account will be worth.

```
PROCEDURE regular_deposits
 0000 100  REM Compute value of regular deposits
 0027 110  PRINT "This program demonstrates the value of saving!"
 005C 120  PRINT
 0061 130  PRINT "How much can you save each month";
 0089 140  INPUT r
 0091 150  PRINT "What is the present interst rate";
 00B9 160  INPUT i
 00C1 170  n=12
 00CC 180  PRINT "How many years are you going to save";
 00F8 190  INPUT y
 0100 200  i=i/n/100
 0113 210  t=r*((1+i)**(n*y)-1)/i
 0136 220  PRINT "You will have $"; INT(t*100+.5)/100; " in the bank!"
 0170 230  PRINT
 0175 240  PRINT "Another Scenario";
 018D 250  INPUT a$
 0195 260  IF a$="y" THEN 120
 01A8 270  IF a$="Y" THEN 120
 01BB 280  END
```

Regular_Deposits09

```
PROCEDURE regular_deposits09
 0000
 0001      (* This procedure computes the value of regular deposits
 0039
 003A      DIM totalsaved,deposit,interest,timesperyear:REAL
 004D      DIM years:INTEGER
 0054      DIM answer:STRING[1]
 0060
 0061 10   PRINT
 0066      PRINT "This program demonstrates the value of regular saving."
 00A0      PRINT
 00A2      INPUT "How much can you save each month? ",deposit
 00CC      INPUT "What is the present interest rate? ",interest
 00F7      INPUT "How many years do you hope to save? ",years
 0123
 0124      timesperyear:=12
 012C      interest:=interest/timesperyear/100
 013C      totalsaved:=deposit*((1+interest)**(timesperyear*years)-1)/
                                                                        i
 015D
 015E      PRINT
 0160      PRINT "You will have $"; INT(totalsaved*100+.5)/100; " in the bank!"
 0197      PRINT
 0199
 019A      INPUT "Another Scenario? ",answer
 01B4
```

```
01B5      IF answer="Y" OR answer="y" THEN
01CA        RUN regular_deposits09
01CE
01CF      ENDIF
01D1      END
```

A BASIC09 PROCEDURE BORN OF NECESSITY

Here is another BASIC09 procedure that may come in handy for you some day. One night, after a long, hard evening filled with writers block and a general lack of creativity, I saved my efforts and went to bed. The next evening I tried to find out how many words I had written by running the wc — word count — utility. The utility reported zero words in the file.

When I tried to list my article, OS-9 returned a CRC error. When I tried to copy it to another file, I got the same result. I kept trying. Nothing worked.

I didn't have the energy to rewrite 3,000 words. Further, my deadline had arrived and I didn't have the time. I had to find a way to recover my work. I used a disk edit utility I own to dump the sectors in the bad file directly from the disk.

As it turned out, the first sector of the file contained garbage and OS-9 refused to read it. Since everything else in the file seemed to be all right, the answer was to skip the first sector. The BASIC09 procedure below did the job.

```
PROCEDURE recover
(* A way to skip a bad sector *)

DIM char,path,newpath:BYTE

OPEN #path,"KISS.temp":READ
CREATE #newpath,"KISS.recovered":WRITE

SEEK #path,257

WHILE NOT(EOF(#path)) DO
  GET #path,char
  PUT #newpath,char
  PUT #1,char
ENDWHILE

END
```

You can also use the PROCEDURE Recover if the bad sector is located somewhere in the middle of the file. To do this, you need to LIST the original file to another file. The LIST command should work up to the point where you hit the bad sector.

At this point you can count the number of bytes you have listed into the recovery file. Add 256 to this number and change the SEEK statement in the PROCEDURE Recover to take you past the bad sector. After you make this change, run the PROCEDURE Recover.

Then, merge the file created with "recover" with the file you LISTed earlier into a new file. After you complete these steps, you will only need to rewrite the copy that was stored in the bad sector. Rewriting 256 characters is a whole lot better than rewriting 3,000 words.

THREE BASIC09 FILTERS

We have included the source code for three BASIC09 procedures that can be run as filters. They get their input from the standard input path — path number zero. They write their output to the standard output path — path number one.

Each of the procedures below can be PACKed into your current execution directory, usually /d0/CMDS, and run as commands. OS-9 will automatically load RunB and execute it when you ask for these procedures by name. RunB must be in memory or stored in your current execution directory when you run these commands, however.

PROCEDURE NEWSTRIP

NewStrip will remove all control characters from a file except for carriage returns and line feeds. It sometimes comes in handy when you need to edit a file you have downloaded from a bulletin board with your screen editor. Use the following command line:

OS9: list anyfile ! newstrip >anyfileminuscontrols

```
PROCEDURE NewStrip
(* A program to strip off all control characters in a file *)
(* except CR's, LF's and DEL's *)

(* modified by Dale L. Puckett to get its input and output *)
(* from standard input and standard output *)

DIM count,Control:REAL
DIM CHAR:BYTE
DIM InPath,OutPath,ErrorPath:BYTE

InPath:=0
OutPath:=1
ErrorPath:=2

Control=0 \count=0

ON ERROR GOTO 10

WHILE EOF(#InPath)=FALSE DO
GET #InPath,CHAR
```

```
count=count+1
IF CHAR<$20 OR CHAR=$7F THEN
IF CHAR=$0D OR CHAR=$0A THEN
PUT #OutPath,CHAR
ELSE
Control=Control+1
ENDIF
ELSE
PUT #OutPath,CHAR
ENDIF
ENDWHILE

10 PRINT #ErrorPath
PRINT #ErrorPath,"Total number of characters = "; count
PRINT #ErrorPath,Control; " control characters were stripped from file."
PRINT #ErrorPath
END
```

THE PROCEDURE NEWHEXDUMP

NewHexDump takes its input from the standard input path. It sends its output to the standard output path. All characters are written in hexadecimal. This lets you see control characters that may be imbedded in the file. The program also counts the number of control characters in the file and the number of carriage returns and line feeds. Use this command line:

OS9: list /d0/cmds/dir ! newhexdump <ENTER>

```
PROCEDURE New_Hex_Dump
(* Program prints a structured hexadecimal dump of all *)
(* control and ascii characters in a file *)

DIM count,Control,cr_lf_count:REAL
count=0 \Control=0 \cr_lf_count=0

DIM char_blocks_per_line:INTEGER
char_blocks_per_line=0

DIM InPath,OutPath,ErrorPath:BYTE
InPath:=0 \OutPath:=1 \ErrorPath:=2

DIM CHAR:BYTE

DIM Blank:STRING[1]
Blank=" "

ON ERROR GOTO 10

PRINT #OutPath

WHILE EOF(#InPath)=FALSE DO
GET #InPath,CHAR
char_blocks_per_line=char_blocks_per_line+1
count=count+1

IF CHAR<$20 OR CHAR=$7F THEN
IF CHAR=$0D OR CHAR=$0A THEN
cr_lf_count=cr_lf_count+1
ENDIF
PRINT #OutPath USING "'[',H2,']',S1",CHAR,Blank;
```

```
Control=Control+1
ELSE
PRINT #OutPath USING "'[',H2,']',S1",CHAR,Blank;
ENDIF

IF char_blocks_per_line>=16 THEN
PRINT #OutPath
char_blocks_per_line=0
ENDIF

ENDWHILE

10 PRINT #ErrorPath,Blank
PRINT #ErrorPath
PRINT #ErrorPath,"Total number of characters = "; count
PRINT #ErrorPath,"Found "; Control; " control characters in file."
PRINT #ErrorPath,"Of which "; cr_lf_count; " were  <CR> or <LF> "

END
```

THE PROCEDURE SPLITWORDS

The PROCEDURE Splitwords reads a text file from the standard input path. It prints a list of words in the file, one to a line, on the standard output path. Since it is a filter you can pipe its input from the output of the OS-9 LIST utility. It is also possible to pipe its output to the input of another tool like a SORT utility. These command lines will work:

OS9: list yourstory ! splitwords ! sort <ENTER>
OS9: list ashortstory ! splitwords <ENTER>

```
PROCEDURE splitwords

DIM char:BYTE
DIM gotone:BOOLEAN
DIM inpath,outpath,errpath:INTEGER
ON ERROR GOTO 100

inpath:=0
outpath:=1
errpath:=2
gotone:=FALSE

LOOP
GET #inpath,char
IF gotone THEN
IF char=32 OR chr=9 OR chr=13 THEN
gotone:=FALSE
WRITE #outpath
ELSE
PRINT #outpath,CHR$(char);
ENDIF
ELSE
IF chr=32 OR chr=9 OR chr=13 THEN
ELSE
gotone:=TRUE
PRINT #outpath,CHR$(char);
ENDIF
ENDIF
```

```
ENDLOOP
BYE
100 (* We may have  reached the end of file *)
DIM errnum:INTEGER
errnum=ERR
IF errnum=211 THEN
BYE
ELSE
ON ERROR
PRINT #errpath,"Error number: "; errnum
BYE
ENDIF
```

C COMPILER

As soon as I received my copy of Radio Shack's new C Compiler from Microware, I went to work and compiled my first program.

```
main()
{
    int sum, x, y;

    x = 20;
    y = 30;
    sum = x + y;

    printf("This is my first 'C' program.\n");
    printf("The sum of %d and %d = %d",x,y,sum);

}
```

Granted, the program doesn't do too much. But it compiled perfectly and ran the first time. It was quite a sight to watch. To compile the program, I typed the line:

OS9:cc CTest.c

Step by step, my Color Computer went through the motions needed to compile a C program. In several minutes it ran these programs.

c.prep (a macro pre-processor)
c.pass1 (OS-9 Level I systems)
c.pass2 (require two passes)
c.opt (the assembly code is optimized)
c.asm (an assembled by a relocating assembler)
cc.link (an finally linked by a linkage editor)

C is not interactive like BASIC09, but it sure is an effective package. Written by James McCosh, author of several 6809 C compilers, and fine tuned to OS-9 by the programmers at Microware, this language is implemented almost exactly as described in "The C Programming Language" by Kernighan and Ritche.

Bit fields are the only thing missing. Other differences between the C description in K & R and the Color Computer C can be counted with the fingers on one hand. And, they all reflect parts of C that are obsolete, or constraints imposed by memory limitations of the Color Computer.

C is not one of the most elegant languages around, but it gives you a solution to a lot of different problems. It is sort of a high-level assembly language.

One of the things C does have going for it is the fact that its code is highly transportable. You can write a program on the Color Computer and carry it over to an IBM PC for, example. The power of the C language can be attributed to the fact that most C programmers use libraries, written in C, which can be adapted to any environment.

One real plus for the Radio Shack C from Microware is the fact that it supports almost all system calls for both OS-9 and UNIX. This means you can write a C program on the Color Computer, port the source code to a 68000 computer running UNIX, compile it there and run it. Microware made this possible by using UNIX names for system functions, even though the same OS-9 function might have a different name.

Sometimes, there are UNIX functions that do not have an exact OS-9 equivalent. Because of this, Microware gave you a library function to simulate the UNIX function. Finally, when there is an OS-9 function that does not have a UNIX equivalent, OS-9 names are used.

This C also has an optional profiler which can be used to determine how many times a function is executed when a program is run. This means you can identify the most frequently used functions and study them in an attempt to find a more efficient algorithm.

SAMPLE PROGRAMS

As we did with BASIC09, we will show you several C programs. Study them and see how they work. You'll be up to speed in no time.

The first several programs were written by Tim Harris, a computer science student at the University of Iowa in Ames. Since they all function as filters, they are a valuable addition to your OS-9 tool kit.

'D' is a special directory utility that lists filenames one line at a time. Its output can be piped into the input of DL, the delete utility in Chapter 18. You'll also find a word count utility and another version of the CAT command.

The last group of C programs were written by Bill Ball, a U. S. Coast Guard public affairs specialist, so that he could analyze his writing. Since Ball believes in Readability, he wrote a filter that counts the number of characters, words and sentences in a story. It reports the average number of letters in a word and the average number of words in a sentence. (Note that Ball's definition of a "word" is a bit non-standard, so his word count will not agree with the "wc" utilities.) His program even shows you how to deal with terminal cursor control codes in C. Enjoy!

wc

```
/* wc utility  ver 2.0 - word, line and char counting          */
/* modeled after UNIX wc utility                                */
/* CoCo OS-9  ver 01.00.00        Microware C Compiler          */
/*                                                              */
/* Contributed to the OS-9 Tour Guide by:                       */
/* Tim Harris (c) 1984                                          */
/* 651 Pammel Court                                             */
/* Ames, Iowa 50010                                             */
/*                                                              */
/* Options:     -l : line count only                            */
/*              -w : word count only                            */
/*              -c : char count only                            */
/*              -? : generate help message                      */
/*                                                              */
/* Uses stdin and stdout, may be redirected and piped           */
/* Calls:                                                       */
/*        wc < file       coundts lines, words and chars        */
/*                        in file                               */
/*        wc -l < file    counts lines only in file             */
/*        d ! wc -l       reports number of files in dir        */
/*   spint pat <file ! wc.c -l reports occurances of "pat"      */
/*                               in the file                    */
/*        wc -?           generates help message                */
/*                                                              */

#include <stdio.h>
#define TRUE 1
#define FALSE 0

main(argc,argv)
int argc;
char **argv;
{
    int c,nl,nw,nc,inword;
    int lon,con,won;
    char *s;

    inword = FALSE;
    nl = nc = nw = 0;
    lon = won = con = TRUE; /* default all on */

    while (--argc > 0 && (*++argv)[0] == '-')
       for (s = argv[0]+1; *s != '\0'; s++)
          switch (*s) {
          case 'l':
             won = con = FALSE;
             break;
          case 'w':
             lon = con = FALSE;
             break;
          case 'c':
             won = lon = FALSE;
```

```
            break;
         case '?':
            help();
            break;
         default:
            printf("wc : illegal option %c\n",*s);
            help();
            break;
         }

   while ((c = getchar()) != EOF){
      ++nc;
      if (c == '\n')
         ++nl;
      if (c == ' ' || c == '\n' || c == '\t')
         inword = FALSE;
      else if (inword == FALSE){
         inword = TRUE;
         ++nw;
      }
   }
   if (lon)
      printf(" %d",nl);
   if (won)
      printf(" %d",nw);
   if (con)
      printf(" %d",nc);
   printf("\n\n");
}

help()
{
     printf("wc      : word count utility\n");
     printf("options: -l   line count only\n");
     printf("         -w   word count only\n");
     printf("         -c   char count only\n");
     printf("         -?   help message \n");
     printf("call    : wc [option] \n");
     exit(0);
}
```

PR

```
/* pr utility - print with pagnation & numbering */
/* CoCo OS-9   C Compiler 01.00.00        */
/* Contributed to the OS-9 Tour Guide by */
/* Tim Harris (c) 1984                   */
/* Options : -n : line numbering         */

#include <stdio.h>
#define NULL 0
#define MAXLINE 256
#define PAGLEN  63

main(argc,argv)
int argc;
char *argv[];
{
    FILE *fp, *fopen();
    int linenum=1, number=0;
    char line[MAXLINE];

    if (argc > 3  || argc == 1)
       error ("pr: improper arguments",NULL);
    if (argc == 2){
```

```
      if ((fp=fopen(argv[1],"r")) == NULL)
         error ("pr: can't open %s",argv[1]);
   }
   else {
      if (argv[1][0] == '-' && argv[1][1] == 'n')
         number = 1;
      else
         error ("pr: illegal option %c",argv[1][1]);
      if ((fp=fopen(argv[2],"r")) == NULL)
         error ("pr: can't open %s",argv[2]);
   }

   while (fgets(line,MAXLINE,fp) > NULL){
      if (number)
         printf("%05d ",linenum);
      else
         printf("      ");
      printf("%s",line);
      ++linenum;
      if (linenum > PAGLEN){
         linenum = 1;
         printf("\n\n\n");
      }
   }
   while (linenum <= PAGLEN){
      ++linenum;
      printf("\n");
   }
   printf("\n\n\n");
   fclose (fp);
   exit(0);
}
```

CAT

```
/* cat utility - concatenate files                                   */
/* modeled after UNIX cat utility                                    */
/* CoCo OS-9  ver 01.00.00       Microware C Compiler                */
/*                                                                   */
/* Contributed to the OS-9 Tour Guide by:                            */
/*                                                                   */
/* (c) 1984                      Tim Harris                          */
/*                               651 Pammel Court                    */
/*                               Ames, Iowa 50010                    */
/*                                                                   */
/* Copies files from stdin or files to stdout, may use               */
/* redirection on output to a file                                   */
/* Call:  cat [file]....[file]                                       */
/*                                                                   */
#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
    FILE *fp, *fopen();

    if (argc == 1)       /* no args - use stdin */
       filecopy (stdin);
    else
       while (--argc > 0)
```

```
            if ((fp = fopen(*++argv,"r")) == NULL){
               printf("cat: can't open %s\n",*argv);
               exit(1);
            } else {
               filecopy(fp);
               fclose(fp);
            }
}

filecopy(fp) /* copy file fp to stdout */
FILE *fp;
{
    int c;
    while ((c = getc(fp)) != EOF)
       putc(c,stdout);
}
```

SPINT

```
/* spint utility - Search and PrINT file for expression      */
/* modeled after UNIX grep utility                           */
/* CoCo OS-9   ver 01.00.00        Microware C Compiler      */
/*                                                           */
/* Copyright (c) 1984              Tim Harris                */
/*                                 651 Pammel Court          */
/*                                 Ames, Iowa 50010          */
/*                                                           */
/* Options:    -x  lines except those with match             */
/*             -n  line numbers printed                      */
/*             -?  help message generated                    */
/* Wildcard:    ?  single char wildcard                      */
/*                                                           */
/* Uses stdin and stdout can be redirected and piped         */
/* Calls:                                                    */
/*        spint the <file  lists lines containing "the"      */
/*        spint -x the<file lists lines without "the"        */
/*        spint the <file ! wc -l counts occurances of       */
/*                                word "the" in file         */
/*        spint -n b?t <file  lists lines with "but",        */
/*                            "bat", ....                    */

#include <stdio.h>
#define MAXLINE 100
#define TRUE 1
#define FALSE 0

main(argc,argv)
int argc;
char *argv[];
{
     char line[MAXLINE],*s;
     long lineno=0;
     int except=FALSE,number=FALSE;

     pflinit();     /* initialize to print long numbers */
     while (--argc > 0 && (*++argv)[0] == '-')
          for(s=argv[0]+1; *s!='\0'; s++)
               switch (*s){
               case 'x':
                    except = TRUE;
                    break;
```

```
                case 'n':
                     number = TRUE;
                     break;
                case '?':
                     argc=0;
                     break;
                default:
                     printf("spint: illegal option %c\n",*s);
                     argc=0;
                     break;
                }
     if (argc != 1){
          printf("Spint: Search and PrINT \n");
          printf("Options: -x exception \n");
          printf("         -n line numbering\n");
          printf("         -? help message\n");
          printf("Wildcard: ? single char wildcard\n");
          printf("Call: spint [-x -n -?] pattern \n");
          exit(0);
     }

     while (gets(line)!= NULL){
          lineno++;
          if ((index(line,*argv) != FALSE) != except) {
               if (number)
                    printf("%ld: ",lineno);
               printf("%s\n",line);
          }
     }
}

index(s,p)
char s[],p[];
{
     int i,j,k;

     for(i=0; s[i]!= '\0'; i++){
          for(j=i,k=0; p[k]!='\0'&&(s[j]==p[k] || p[k]=='?');j++,k++)
               ;
          if (p[k] == '\0')
               return(TRUE);
     }
     return(FALSE);
}
```

UPLOW

```
/* uplow utility - convert text to all upper or lower case   */
/* CoCo OS-9 ver 01.00.00          Microware C Compiler      */
/* Copyright (c) 1984              Tim Harris                */
/*                                 651 Pammel Court          */
/*                                 Ames, Iowa 50010          */
/*                                                           */
/* Options : -u convert to upper case                        */
/*           -l convert to lower case (default)              */
/*           -? generate help message                        */
/*                                                           */
/* Uses stdin and stdout and may be redirected or piped      */
/* Call    : uplow [option]                                  */

#include <stdio.h>
```

```
#include <ctype.h>
#define FALSE 0

main(argc,argv)
int argc;
char *argv[];
{
     char *s;

     if (argc == 1)
          makelower();
     else{
          while (--argc > 0 && (*++argv)[0] == '-'){
               for (s = argv[0]+1; *s != '\0'; s++)
                    switch(*s) {
                    case 'l':
                         makelower();
                         break;
                    case 'u':
                         makeupper();
                         break;
                    case '?':
                         help();
                         break;
                    default:
                         printf("uplow: illegal option %c\n",*s);
                         help();
                         break;
                    }
          }
     }
}

makeupper()
{
     int c,i;

     while ((c = getchar()) != EOF){
          if (( i = isalpha(c)) != FALSE)
               if ((i = islower(c)) != FALSE)
                   c = _toupper(c);
          putchar(c);
     }
}

makelower()
{
     int c,i;

     while ((c = getchar()) != EOF){
          if (( i = isalpha(c)) != FALSE)
               if (( i = isupper(c)) != FALSE)
                    c = _tolower(c);
          putchar(c);
     }
}

help()
{
     printf("uplow    : convert text to all upper or lower\n");
     printf("options  : -l make lower case (default)\n");
     printf("           -u make upper case\n");
```

```
    printf("          -? help message\n");
    printf("call     : uplow [option]\n");
    exit(0);
}
```

D

```
/* d utility 01.20 : directory with pattern matching           */
/*                   lists current directory with one          */
/*                   entry per line.                           */
/*                                                             */
/* CoCo OS-9 v01.00.00        Microware C Compiler             */
/*                                                             */
/* Copyright (c) 1984         Tim Harris                       */
/*                            651 Pammel Court                 */
/*                            Ames, Iowa 50010                 */
/*                                                             */
/* uses stdout for output so it may be piped or redirected     */
/* can be used with other utilities, i.e., dl (delete)         */
/*                                                             */
/* Options:     -w       wild card matching                    */
/*                       ? - single char wildcard              */
/*                       * - multiple char wildcard            */
/*              -?       help message                          */
/*                                                             */
/* Calls:       d        lists current directory               */
/*              d -w *.c lists files ending in '.c'            */
/*              d -?     generates help message                */
/*         d -w c*       lists files beginning with 'c'        */
/*         d -w *c*      lists files with a 'c' in them        */
/*         d -w a?c      lists files abc,aec,a.c,...           */
/*         d -w ????     lists files with four chars           */
/*                                                             */
/* Calls with other utilities:                                 */
/*   d -w *.c ! dl -z    deletes all files ending in '.c'      */
/*   d -w ??? ! wc -l    counts number of 3 char filenames     */
/*   d ! sort            prints sorted directory               */
/*                                                             */

#include <stdio.h>
#include <ctype.h>
#define DREAD 129
#define ENTSIZ 32
#define TRUE 1
#define FALSE 0

/* Types for pattype */
#define REG 0       /* regular exact match w/ ? wildcard */
#define BOL 1       /* match at Beginning Of Line */
#define EOL 2       /* match at End Of Line */
#define MID 3       /* match in MIDdle of line */

char dname[2] = {'.','\0'};
int pattype = REG;        /* default to exact or ? match */
char pat[29];

main(argc,argv)
int argc;
char *argv[];
{
```

```
     char c,fname[30],entry[32],*s;
     int i,dp,woption=FALSE;

     while (--argc > 0 && (*++argv)[0] == '-')
          for (s=argv[0]+1;*s!='\0';s++)
               switch(*s){
               case 'w':
                    woption = TRUE;
                    getpat(*++argv);
                    break;
               case '?':
                    help();
               default:
                    printf(" d: illegal option %c\n",*s);
                    exit(0);
               }

     if ((dp=open(dname,DREAD))==-1){
          printf(" Can't open default directory\n");
          exit(0);
     }

     while ((read(dp,entry,ENTSIZ))!=NULL){
          if (entry[0]!=0){
               i=-1;
               do{
                    c=entry[++i];
                    fname[i]=toascii(c);
               }while(isascii(c)!=FALSE && i<=29);
               fname[++i]='\0';
               if (fname[0]!='.')
                   if (woption){
                         if ((isin(fname))!= -1)
                             puts(fname);
                   }else
                         puts(fname);
          }
     }
     close(dp);
}

help()
{
     printf("\n d: directory utility\n");
     printf(" lists current data directory, one entry per line\n");
     printf(" options: -w     wildcard matching\n");
     printf("                 ? - single char wildcard\n");
     printf("                 * - multichar wildcard\n");
     printf("          -?     help message\n");
     printf(" call:    d <-? || -w pattern>\n\n");
     exit(0);
}

getpat(s)
char s[];
{
     int ln,i,j=0;

     ln=strlen(s);
     ln--;
     if (s[0]=='*' && s[ln]=='*')
          pattype=MID;
     else{
          if(s[0]=='*')
```

```
                pattype=EOL;
          else
                if(s[ln]=='*')
                      pattype=BOL;
     }
     for(i=0;s[i]!='\0';i++)
          if(s[i]!='*')
                pat[j++]=s[i];
     pat[j]='\0';
}

isin(s)
char s[];
{
     int i,j,k,ln,pl;

     switch(pattype){
     case REG:
          for(i=0;s[i]!='\0'&&(s[i]==pat[i] || pat[i]=='?');i++)
                ;
          if(pat[i]=='\0' && s[i]=='\0')
                return(i);
          else
                return(-1);
     case BOL:
          for(i=0;pat[i]!='\0'&&(s[i]==pat[i] || pat[i]=='?');i++)
                ;
          if(pat[i]=='\0')
                return(i);
          else
                return(-1);
     case EOL:
          pl=strlen(pat);
          ln=strlen(s);
          for(j=ln-pl,k=0;pat[k]!='\0'&&(s[j]==pat[k] || pat[k]=='?');j++,k++)
                ;
          if(pat[k]=='\0' && s[j]=='\0')
                return(j);
          else
                return(-1);
     case MID:
          for(i=0;s[i]!='\0';i++){
                for(j=i,k=0;pat[k]!='\0'&&(s[j]==pat[k] || pat[k]=='?');j++,k++)
                      ;
                if(pat[k]=='\0')
                      return(i);
          }
          return(-1);
     }
}
```

F

```
/* f utility:  optional formatter for DynaStar */
/* formats with tm=6; bm=60; pl=66; lm=8       */
/* has optional page numbering (use -n)        */
/* has no problems with double spacing         */
/* sends to stdout so use redirection for /p   */
/*                                             */
/* CoCo OS-9    C-Compiler 01.00.00            */
/* (c) 1984 Tim Harris                         */
/* Call:                                       */
/*   f <-n><filename> (> redirection)          */
```

```
#include <stdio.h>
#define MAXLINE 80
#define PAGLEN 54

main(argc,argv)
int argc;
char *argv[];
{
     FILE *fp,*fopen();
     int linenum=1, number=0, pageno=1;
     char line[MAXLINE];

     if (argc>3 || argc==1)
          error ("f: improper arguments",NULL);
     if (argc ==2){
          if ((fp=fopen(argv[1],"r"))==NULL)
               error ("f: can't open %s",argv[1]);
     }
     else{
          if (argv[1][0]=='-' && argv[1][1]=='n')
               number=1;
          else
               error ("f: illegal option %c",argv[1][1]);
          if ((fp=fopen(argv[2],"r"))== NULL)
               error ("f: can't open %s",argv[2]);
     }
     printf("\n\n\n\n\n\n");
     while (fgets(line,MAXLINE,fp)!=NULL){
          printf("        %s",line);
          ++linenum;
          if (linenum>PAGLEN){
               linenum=1;
               printf("\n\n\n");
               if (number)
                    printf("                              %d\n\n\n\n\n\n\n\n\n\",pageno++);
               else
                    printf("\n\n\n\n\n\n\n\n\n");
          }
     }
     while (linenum <= PAGLEN){
          ++linenum;
          printf("\n");
     }
     printf("\n\n\n");
     if (number)
          printf("                                                  %d\n\n\n",pageno++);
     else
          printf("\n\n\n");
     fclose(fp);
}

error(s1,s2)
char *s1,*s2;
{
     printf(s1,s2);
     printf("/n");
     exit(1);
}
```

frep - file report utility for ascii files

shows:
- o number of characters, words, sentences [.!?:;]
- o average number of characters per word, words per sentence, characters per sentence
- o largest word, sentence
- o number of lines, pages

Although written for the Coco with the PBJ WordPak, this program can be modified to run on any terminal. It's written in "standard" C, if there is such a thing.

Compile it using the Microware/Tandy C-compiler:

```
cc1 frep.c -f=frep
```

Frep is fairly versatile. It is fastest when the report is sent through the standard output. On a 1Mhz Coco, frep will read a 10-page, single-spaced file in 65 seconds. If you send the report to the screen it takes three times as long.

Syntax:

Example: `frep <file>`

Reads file, sending a report through the standard output. You can redirect the report to the printer with:

Example: `frep <file> >/p`

or a disk file with:

Example: `frep <file> >/<path>`

You can also send the report to the screen with a rolling number output. This demonstrates the use of cursor control in C. Use the [-t] option.

Example: `frep -t <file>`

Don't redirect the output with the -t option, or you'll get the cursor control codes sent to the file or printer.

Frep will accept multiple files on the command line. This will give you a series of reports, one for each file.

Example: `frep <file1> <file2> <file3>`

You may also "pipe" the output of the list command through frep to get a cumulative report instead of separate ones.

Example: `list <file> <file> <file> ! frep`

This also works with the [-t] option.

Finally, if you forget what the options are, use:

Example: `frep -?`

I hope you have fun with this. I know I did. I only wish

```
     there were more commands and utilities availble in C for
     the CoCo, in the public domain, and in source form.  I
     know I'm going to contribute.  I'm excited and still
     learning.  How about you?

     version 1.0 created 05/02/84
     version 2.0 updated 07/23/84 standard input/multiple files
     version 3.0 updated 08/10/84 terminal codes/rolling numbers

                    by:  William H. Ball
                         9007B Saratoga Dr.
                         Indianapolis, IN 46236
                         (317) 897-2661
*/

#include <stdio.h>
#define PGLN 55              /* number of lines per printed page     */
#define CLEARS 2             /* clear screen/home cursor    ^B       */
                             /* (12) for Tandy                       */
#define CURSOR 20            /* ^T = lead-in for x/y control         */
                             /* (2) for Tandy                        */
#define OFFSET 31            /* offset for row,col placement         */
                             /* (32) for Tandy                       */
#define BACK 8               /* backspace ^H same for Tandy          */
int term = 0;                /* report to terminal switch (1) = yes */

main(argc, argv)
int argc;
char *argv[];
{
     int i;
     FILE *fp;
     while (argc > 1 && argv[1][0] == '-') {
          switch (argv[1][1]) {
          case '?': /* shows usage */
               usage();
               break;
          case 't': /*  send report to terminal */
               term = 1;
               break;
          default:
               fprintf(stderr, "%s: unknown arg %s\n",
                    argv[0], argv[1]);
               usage();
               exit(1);
               break;
          }
          argc--;
          argv++;
     }
if (argc == 1)
          frep(stdin);
else
     for (i = 1; i < argc; i++)
          if ((fp=fopen(argv[i], "r")) == NULL) {
               fprintf(stderr, "%s: can't open %s\n",
                    argv[0], argv[i]);
               exit(1);
          } else {
               frep(fp);
               fclose(fp);
          }
```

```
    exit(0);
}
frep(fp)  /* actual report function */
    FILE *fp;
{
    float avgchar, /*   avg # of chars per word        */
          avgword,  /*   avg # of words per sentence    */
          avgsen,   /*   avg # of chars per sentence    */
          totchars, /*   total # of chars in file       */
          totwords, /*   total # of words in file       */
          totsents; /*   total # of sentences in file   */
    int   longword, /*   # of chars in longest word     */
          longsent, /*   # of words in longest sent     */
          pages,    /*   # of pages in file             */
          remain,   /*   remainder for modulo divide    */
          lines,    /*   #lines in a file               */
          wordsin,  /*   # of words in a sentence       */
          charsin,  /*   # of chars in a word           */
          c;        /*   character in text stream       */

    /* initialize variables */
   totwords = totchars = totsents = longword = longsent = avgchar =
   avgword = avgsen = pages = remain = lines = wordsin = charsin = 0;

    if ( term == 1 ){
          clrtrm(); /* clear screen/home cursor */
          report(); /* print blank report page */
          term = 1;
    }
    /* main algorithm begins here */
          while ((c = getc(fp)) != EOF) {
                if(c == ' ') {
                      ++wordsin;
                      totchars = totchars + charsin;
                      if(charsin > longword) { longword = charsin; }
                      charsin = 0;
                }
                else if(c == '.' || c == '?' ||
                         c == '!' || c == ':' || c == ';') {
                      ++charsin;
                      ++wordsin;
                      totwords = totwords + wordsin;
                      ++totsents;
                      totchars = totchars + charsin;
                      if(wordsin > longsent) { longsent = wordsin; }
                      if(charsin > longword) { longword = charsin; }
                      wordsin = 0;
                      charsin = 0;

                      if ( term == 1 ){
                                pffinit();
                                set_cur(3, 48);
                                printf("%.0f", totchars);
                                set_cur(5, 48);
                                printf("%.0f", totwords);
                                set_cur(6, 48);
                                printf("%d", longword);
                                set_cur(10, 48);
                                printf("%.0f", totsents);
                                set_cur(11, 48);
                                printf("%d", longsent);
                      }
```

```
                }
                else if(c == '\n') { ++lines; }
                else ++charsin;
            }
/* fill in rest of screen after computing averages */
                    if ( term == 1 ){
                        avgchar = totchars / totwords;
                            set_cur(7, 48);
                            printf("%.2f", avgchar);

                        avgword = totwords / totsents;
                            set_cur(8, 48);
                            printf("%.2f", avgword);

                        avgsen = totchars / totsents;
                            set_cur(12, 48);
                            printf("%.2f", avgsen);

                        pages = lines / PGLN;
                            remain = lines % PGLN;
                                if(remain >= 1) { ++pages; }
                                    set_cur(14, 48);
                                    printf("%d",pages);
                            set_cur(15, 48);
                            printf("%d", lines);
                            set_cur(24,1); /* clean finish */
                            putchar(BACK); /* prevent scroll */
                    }/* inserted for ( term ) */
                    if ( term == 0 ){
    pffinit(); /* this statement is necessary to print all     */
               /* floats or doubles variables. See pg. 4-20    */
               /* of the Tandy C-compiler manual               */

    printf("\n\nFile report: \n\n");
    printf("   Number of characters in file: %.0f\n", totchars);
    printf("    Number of words in the file: %.0f\n", totwords);
    printf("Number of chars in longest word: %d\n", longword);

        avgchar = totchars / totwords;
    printf("  Avg word length in characters: %.2f\n\n", avgchar);

        avgword = totwords / totsents;
    printf("    Avg # of words per sentence: %.2f\n", avgword);
    printf("     # of sentences in the file: %.0f\n", totsents);
    printf(" # of words in longest sentence: %d\n", longsent);

        avgsen = totchars / totsents;
    printf(" # of chars in average sentence: %.2f\n\n", avgsen);

                        pages = lines / PGLN;
                            remain = lines % PGLN;
                                if(remain >= 1) { ++pages; }

        printf("         Number of pages in file: %d\n",pages);
        printf("         Number of lines in file: %d\n", lines);
                    }
}/* end of frep */

/*usage function - prints usage of the frep utility                    */
usage(){
    printf("Syntax:\n");
    printf("frep <file> (report on single file)\n");
```

```
     printf("list <file> <file> ! frep (cumulative file report)\n");
     printf("frep (stdin)\n");
     printf("frep [-?] prints usage\n");
     printf("frep [-t] sends slow terminal report");
     exit(0);
}

/* set_cur function - performs the cursor placement - needs
 *   two arguments: row, col. CURSOR is lead-in for x,y screen
 *   control.  OFFSET is x,y offset (31,31)=row(0),col(0)
 */
set_cur(row, col)
int row, col;
{
     printf("%c%c%c", CURSOR, row+OFFSET, col+OFFSET);
}

/* report function - prints blank report page set_cur fills in   */
report(){
     printf("File report:\n\n");
     printf("                       number of characters:\n");
     printf("\n                              number of words:\n");
     printf("                              longest word:\n");
     printf("      average number of characters per word:\n");
     printf("       average number of words per sentence:\n");
     printf("\n                          number of sentences:\n");
     printf("                           longest sentence:\n");
     printf(" average number of characters per sentences:\n");
     printf("\n                              number of pages:\n");
     printf("                            number of lines:\n");
}

/* clrtrm function - clears screen/homes cursor */
clrtrm(){
     putchar(CLEARS);
}
```

OS-9 PASCAL

Information about the Radio Shack version of OS-9 Pascal is sketchy as this book goes to print. However, we wanted to give you a brief overview based on our experiences with the Microware original version of Pascal.

This Pascal is an example of the new breed of software tools that were originally developed and perfected on large computer systems. In addition to keeping all the features of the larger mainframes, Microware has added features that increase your productivity as a programmer and improve the reliability of your programs.

The Radio Shack Pascal produces an intermediate code like BASIC09. The Pascal intermediate code is called p-code. This p-code can be run on three individual interpreters that come with the system. Then, once a program is debugged, you can use another program to translate it into native 6809 code. You literally have the best of both worlds.

The OS-9 Pascal package includes:

PASCALN — A p-code interpreter
PASCALT — A translator that converts p-code to native 6809 code
PASCALS — A virtual memory "swapping" p-code interpreter
PASCALE — A Pascal linkage editor
SUPPORT — A native code run time subroutine package

PASCALN is used to run small to medium size Pascal programs. It is fast and efficient, and typically your programs will run at about one-half the speed of native 6809 code.

PASCALS is used to run small — to very large — Pascal programs using a transparent virtual memory code swapping scheme. You can manage the tradeoffs between memory size and execution speed when you run this program.

PASCALT is used once you have optimized your p-code. When you use native 6809 code, you trade increased memory requirements for programs that run faster. PASCALT does not need to be used on an entire program either. You can just translate the critical routines if you desire.

OS-9 Pascal is a nearly full implementation of the Pascal language defined by the ISO 7185.1 level 0 specification. It supports the BOOLEAN, CHARACTER, INTEGER, and REAL data types.

SUMMARY

In this chapter you have been introduced to three of the high level languages that run in the OS-9 environment. In addition to a brief overview of each one, we have presented several program listings to help you understand BASIC09 and C programming under OS-9.

Part V of our OS-9 tour guide begins in the next chapter. In the next dozen chapters, you will move closer and closer to the pot of gold that awaits you after you learn the inner workings of OS-9.

CHAPTER 20

managing your memory

In this chapter we discuss prudent management of memory.

Your system's main memory (RAM) is a resource that is always too scarce. It can seem that OS-9 has taken memory allocation completely out of your hands, but, in fact, there is a lot you can do about the way memory is used.

REENTRANT MODULES

The most important step to take toward conserving memory is to use reentrant modules whenever possible. It is often worth considerable trouble to make a module reentrant. Reentrancy isn't that rigorous a requirement. A reentrant program must base all its variables off index registers or the DP register. PC relative values and absolute (extended) addresses can only be used as constants. Programs that modify themselves are strictly out of the question.

The modules you are most likely to create will contain programs, either in intermediate code (like BASIC09 I-Code), or in 6809 object code. BASIC09 always creates reentrant code, so do all other higher level languages for OS-9. If you write in assembler, you can do whatever you want, including writing non-reentrant code.

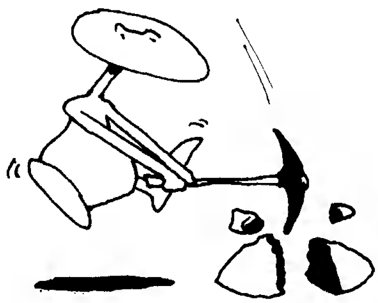
The great advantage of reentrant modules is that any number of processes can share them. Each process must have its own data

storage, but any number of processes can share the program itself. Not having to store a separate copy of the program for each process can save significant amounts of memory. As an example, note that if BASIC09 weren't reentrant, each process using it would start by requiring a bit over 23,000 bytes just for its copy of BASIC09. As it is, most systems can run at least three or four BASIC09 programs concurrently.

Making a module reentrant is the easy part. Making a module general enough that several processes might want to use it concurrently is the real trick. Important system programs like the Editor and BASIC09 have it easy. The designers knew those programs would be heavily used. People writing more specialized programs have to be careful.

It isn't likely that most full-blown programs will be used by several processes unless the system is dedicated to the task performed by the program. However, there are some operations that many programs have in common: formatting output, math, validating input, formatting the screen and handling database files are some examples. If all these functions are built into one module, a program will have to incorporate the entire package if it needs to use any of the functions in the module. If separate modules are built to do each of these operations, some of them might make useful parts of several different programs. OS-9 makes it easy for a program to collect a group of modules. You, the programmer, have to design your system so that feature is useful.

MEMORY FRAGMENTATION



Memory fragmentation is a problem that more serious OS-9 Level One users will have to learn to deal with. Hardware included in each Level Two system (the DAT) makes memory fragmentation irrelevant for them. Memory fragmentation becomes a problem when the available free memory is in so many little pieces that OS-9 can't find enough memory in one block to satisfy a program's request. This is a serious problem, and is dealt with in more depth in the "Over the Rainbow" section. The simple solution to memory fragmentation problems is to kill processes, starting with the ones you can spare most easily, and continuing until there is enough contiguous memory to satisfy you. The MFREE command will tell you how many blocks of free memory you have, and how big they are.

CAREFUL USE OF MEMORY

There are some ways to waste memory that (I think) can only be done intentionally. The best example is the SLEEP command. If SLEEP is given extra memory, it takes it out of circulation. You can be certain SLEEP doesn't run any faster or better with more memory. If you want to waste memory for some reason, use:

SLEEP 10000 #48&

There go 48 pages (12K) of memory out of circulation until the command terminates.

A less obvious way to waste memory is to give a program more memory than it needs. It is easy to allocate extra memory to a program with the "#" shell option. Some programs make good use of extra memory, some don't even notice it. Even the other programs running at the time influence how useful extra memory is. In general, give a program extra memory only if you KNOW it will help. The COPY command is a good example. Everyone knows COPY runs better if it is given extra memory; no question, it does. But just how much better it runs depends on your disk hardware and what else is going on in your system. If you have a hard disk, extra memory makes almost no difference. On the other side of the issue, the assembler really needs extra memory to assemble most programs.

Some programs, like BASIC09 and Asm, make it very clear to you when they need more memory. They don't care how much memory they have, provided it is enough. You can prove this to yourself by running the assembler with 10K and then with 32K. It will run in the same time if it runs at all. Programs like BACKUP and COPY will run faster with extra memory, but how much faster depends on the details of your system. If you use the commands a lot, and care about conservation of memory, it would be worth your time to make some tests.

You can waste memory by writing non-reentrant modules, or by giving programs unneeded memory from the shell without ever writing a line of code. OS-9 is also able to waste memory without any effort on your part. It can make your memory useless by slicing it up too small for your programs. This nasty habit is called memory fragmentation.

If you don't push your system, you probably won't have trouble with memory. Fragmentation only happens when several processes run simultaneously. If you don't do that a lot, you won't see the problem. Similarly, if you don't run several programs at once, you don't care about a program taking up unused space — provided it doesn't keep itself out of that memory. Even reentrancy is only an issue when there is more than one process using a module.

CHANGING THE DEFAULT MEMORY ALLOCATION OF A MODULE

If your system is heavily used, especially by a large group of people, you may want to predetermine the best memory allocations for each program. Microware has taken some guesses at what memory allocations their programs should have, but they were always careful. The programs are usually given the minimum memory necessary for running them.

If you want a program to have a default memory different from

that which Microware assigned, modify that module's header. The header contains a field that determines the amount of memory the program will request. I don't recommend ever decreasing that value, but it is often convenient to increase it. Just use Debug, or any other appropriate tool, to increase the value in the M$Mem field, a two-byte field, twelve bytes from the beginning of the module header.

If you reset modules to your preferred memory size, you won't have to remember the correct figure for optimum performance for each program. Real perfectionists won't be happy with any standard memory allocation for a program. They can still give the program a different memory allocation from the shell command line.

DYNAMIC MEMORY ALLOCATION

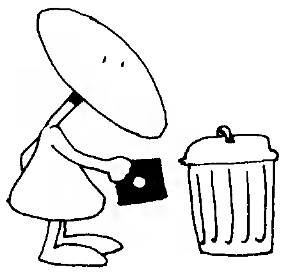
In one case, wasting memory is necessary. If a program, running under OS-9 Level One, wants to call for additional memory, it should do it as early as possible even if it won't use the memory for awhile. Data memory for programs is allocated starting in low memory and working upward. When a program calls for extra memory, it will only get the memory if there is free memory just above the current allocation. If another process has been started, it may well have claimed that memory. So, in the case of programs that increase their memory requirements dynamically, ask for what you need early, and don't let it go until you know you won't need it again.

SUMMARY

In this chapter we covered several of the common pitfalls and problems OS-9 users encounter when dealing with memory allocation. Important concepts are reentrancy, default memory allocation and fragmentation.

CHAPTER 21

managing disk space



Disk space is another kind of memory the OS-9 user gets to worry about. In this chapter we'll discuss how to avoid wasting disk space — or at least know when you're using it inefficiently. We'll also discuss what can slow down access to disk files, and some tricks for recovering damaged or deleted files.

There are two types of files that waste disk space, small files and unexpectedly large files.

USING SPACE EFFICIENTLY

Small files carry a heavy overhead burden compared to their size. The directory entry is a barely noticable 32 bytes. The file descriptor takes a sector (256 bytes). The real overhead is the minimum allocation OS-9 makes for each file. If the minimum allocation is more than one sector, very small files can waste lots of disk space. If you have a choice of many small files or fewer large ones, chose larger files when efficient use of disk space is your first priority.

Enlarging a file under OS-9 is easy. All you have to do is write a byte at a position past the end of a file and the file will be lengthened to include that byte. If you aren't careful, that trick can cause you to allocate more data for a file than it needs. You can protect yourself from inadvertently lengthening a file by avoiding I$Seek on a file when it is open for writing. If you don't use the

I$Seek service request, you won't be able to write a byte way beyond the end of a file without writing all the bytes before it; if you do use I$Seek, be careful.

AIMING FOR SPEED

OS-9 isn't known for its high-speed disk access, though access to a file will seldom be significantly slower than is usual for the computer it's running on. The most common cause for slow access to files is the environment; that is, what other files are being accessed and what processes are running. Another problem that sometimes is important is disk fragmentation.

It takes time to move a disk drive's read/write head from track to track on a disk. Because of this, it is fastest to access a file with the minimum movement of the heads. Before reading a file, OS-9 must read the directory to find the address of its file descriptor, then it must read the file descriptor. The file descriptor contains a list of address/length pairs that describes the location of the file on the disk. In the best case, the file can be read with only two extra reads: one for the directory entry, the other for the file descriptor. If the file had to be scattered around the disk because disk memory was fragmented, there will be several address/length pairs in the file descriptor, and OS-9 will have to keep returning to the File Descriptor to find the address of the next block.

The only way to deal with fragmentation is to copy the contents of the disk, file by file, from one disk to another. Using BACKUP doesn't help because BACKUP makes a mirror image of the disk; each fragmented file will be just the same on the new disk as on the old.

If optimization is very important, a file's directory entry should be placed early in the directory; otherwise several sectors of directory may have to be read before the right entry is found.

This trick is most useful in the execution directory. There are many entries in that directory, and most of the files are small enough that a few extra reads to search through the directory make a noticable difference.

For maximum speed, put your most-used files near the beginning of the directory.

REPAIRING DAMAGE

To recover a damaged or deleted file on disk, you must have a way of reading and modifying selected sectors on disk. Several programs are available to do this, including a program from the OS-9 Users Group and a sample BASIC09 program in the BASIC09 Manual.

If a small file has been erased and you catch it soon enough, you can recover it.

- First, inspect the directory the file came from. You should find a directory entry that has the file name of the deleted file in it minus the first and last letters. The first letter was changed to a $00 to indicate that the file was deleted. The last letter isn't a standard ASCII character because the high order bit in it was set "on" to indicate that it was the last byte in the file name.

- When you have found the right directory entry, change its first byte back to the first character in the file name.

- Now the file will show up in the directory and be readable. Don't write anything on this disk yet! The sectors that make up the deleted file are still marked as free for use. The next file that gets written to the disk may well use an important part of the file you are trying to rescue.

- Copy the file to another disk — not another file on the disk with the deleted file!

- If the previous step doesn't work, either an important sector (the File Descriptor) in the file you are trying to rescue was used by another file, or the file was too long for this method. Give up unless you want to search through all the sectors on your disk, pulling out those that contain parts of the file.

- Put the directory back the way you found it.

- The disk is now usable. You may want to copy the rescued file back to its original location.

This method only works for files that are one extent long. If OS-9 didn't allocate the file in one contiguous block, recovering the file will be much harder. Unfortunately, this means that often you'll have to struggle to recover a file. OS-9 seldom allocates files of more than a few hundred bytes in one extent. You can probably reconstruct small files painlessly, but big files contain large amounts of data in several extents.

USING BRUTE FORCE

If the consequences of losing a file are so dreadful that it's worth hours of your time to recover it, you can retrieve the data the hard way. Like the trick for recovering small files this requires two disk drives. Unlike the trick for small files, it takes lots of time and effort.

This isn't really a trick. It's just a brute force approach to the problem. In essence, we're about to treat the disk with the deleted file as a single file that includes all the sectors on the disk. You look through all the sectors, selecting ones that look like part of the file you want to recover, and build a file including all those sectors. Then, using an editor, you put those sectors into the right order.

It's difficult to recognize a chunk out of the middle of an object module, so lets assume the deleted file contained text. The BASIC09 program "Scavange" runs through each sector on disk /D1. It displays the contents of the sector on the screen with the question "Keep?" If you reply, ' Y ' it will copy the sector to the file "recover" in your data directory. Any other reply, even just a return, will cause the program to go on to the next sector.

```
PROCEDURE Scavange
 0000      DIM sector(256),fsector(256):BYTE
 0015      (* sector is the buffer for put and get *)
 003F      (* fsector is used for data formated for display *)
 0072      DIM diskname:STRING[10] \(* name of disk to recover *)
 009B      DIM recname:STRING[10] \(* name of file for recovered data *)
 00CC      DIM recfile,diskfile:INTEGER \(* file numbers *)
 00E9      DIM yn:STRING[3]
 00F5      DIM done:BOOLEAN
 00FC      DIM i:INTEGER
 0103      recname="Recovery"
 0112      diskname="/D0@"
 011D      done=FALSE
 0123      OPEN #diskfile,diskname:READ
 012F      CREATE #recfile,recname:WRITE
 013B      RUN getsector(diskfile,sector,done)
 014F      WHILE NOT(done) DO
 0159        RUN reformat(sector,fsector)
 0168        RUN display(fsector,yn)
 0177        IF LEFT$(yn,1)="Y" THEN
 0187          PUT #recfile,sector
 0191        ENDIF
 0193        RUN getsector(diskfile,sector,done)
 01A7      ENDWHILE
 01AB      CLOSE #diskfile,#recfile
 01B6      END
```

```
PROCEDURE getsector
 0000      PARAM diskfile:INTEGER
 0007      PARAM sector(256):BYTE
 0013      PARAM done:BOOLEAN
 001A      DIM errno:INTEGER
 0021      ON ERROR GOTO 100
 0027      GET #diskfile,sector
 0031      END
 0033 100  (* expect end-file
 0048      ON ERROR
 004B      errno=ERR
 0051      IF EOF(#diskfile) THEN
 005B        done=TRUE
 0061      ELSE
 0065        PRINT "Error "; errno; "on /D0"
 007C      ENDIF
 007E      END
```

```
PROCEDURE reformat
 0000      PARAM in(256),out(256):BYTE
 0015      DIM i:INTEGER
 001C      FOR i=1 TO 256
 002D        IF in(i)>=ASC(" ") AND in(i)<=ASC("~") OR in(i)=13 THEN
 0054          (* in(i) is a printable character or a <CR> *)
 0082          out(i)=in(i)
 0091        ELSE  \(* in(i) isn't printable *)
 00B0          IF in(i)>=ASC(" ")+128 THEN
 00C4            out(i)=in(i)-128
 00D6          ELSE
 00DA            out(i)=ASC(".")
 00E7          ENDIF
 00E9        ENDIF
 00EB      NEXT i
 00F6      END
```

```
PROCEDURE display
 0000      PARAM fsector(256):BYTE
 000C      PARAM yn:STRING[3]
 0018      DIM work:STRING[256]
 0024      DIM i:INTEGER
 002B      i=1
 0032      work=""
 0039      WHILE i<=256 DO
 0046        IF fsector(i)<>13 THEN
 0055          work=work+CHR$(fsector(i))
 0065        ELSE
 0069          PRINT work
 006E          work=""
 0075        ENDIF
 0077        i=i+1
 0082      ENDWHILE
 0086      PRINT work
 008B      PRINT
 008D      INPUT "keep? ",yn
 009B      IF LEFT$(yn,1)="y" THEN
 00AB        yn="Y"
 00B3      ENDIF
 00B5      END
```

It is easy to reject the last sector in a file by mistake. The last sector in a file will contain the file's last characters, but the rest of it will be filled with junk. To prevent the junk from deceiving you, keep your eye on the beginning of each sector.

If you are certain you have retrieved all you want from a file before Scavange has worked through the entire disk, abort the program with a keyboard interrupt. You'll spend long enough running Scavange without extending the pain!

You won't find a damaged file very often. If you use high-quality diskettes and take good care of them, you may never see a damaged file. The only disks I have had any trouble with are those I

receive in the mail. The Post Office is the great destroyer of diskettes. They (the diskettes) can only stand so much heat, cold, and folding. Even when a diskette makes it through the Postal filter intact, there is a chance for disaster. Your drives may have trouble reading disks written by someone else's drives. Don't give up at the first #244 error. If you can read part of a file, there is a good chance you can get at most of it.

If there is a bad sector somewhere in the file, your best bet is to try to read it several times. You might even try writing a program that copies the file and retries many times if it gets a I/O error. If that trick doesn't work, you'll have to give up on that sector and try to rescue the rest of the file.

HOW TO IGNORE A BAD SECTOR

You can eliminate a bad sector from a file by fussing with the File Descriptor. The block of the file that contains the bad sector must be split into two blocks, with neither containing the bad sector.

Inspect the File Descriptor for the damaged file.

Decrease the byte count FD.SIZ by 256, the size of a sector.

Look through the Segment List for the address/length pair that points to the block containing the damaged sector.

Divide that block into two segments:

> The first having the same address as the original, but a length that only takes it up to the sector before the damaged one.
>
> The second having an address one sector beyond the damaged sector and a length equal to the original length minus one for the damaged sector, minus the length of the first new segment.

Copy the file. It's all right to copy it to another file on the same disk.

Delete the damaged file.

The damaged sector will still be marked as allocated. This will cause the disk to show an error when DCHECK is run on it, but it is easier than returning the file descriptor to its original form before deleting the file, and it prevents the bad sector from being used for another file.

This method sounds complicated. It is. Fortunately, there is an easier way. Skip over the bad sector while reading the file. Write a program that copies the file. When the program gets a read error, it should get the position in the file, round it up to the next multiple of 256 and seek to that position in the file. This trick jumps over any bad sectors.

SUMMARY

In this chapter you learned about clever and efficient use of disk space. You also learned several ways to recover lost and damaged disk files.

CHAPTER 22

building a device descriptor

Device descriptors are OS-9's reference material for I/O devices. In this chapter we will talk about how they are constructed and what they do.

WHY MAKE DEVICE DESCRIPTORS?

There is a device descriptor for every I/O device in an OS-9 system. As the name implies, each descriptor describes the attributes of a device. They each contain a description of the hardware for a device and other information specific to it. A new terminal port or graphics card will need a new device descriptor. Sometimes, just changing the type of terminal attached to a serial port will require some changes to the descriptor for that port.

Device descriptors contain all that OS-9 needs to know about a device. However, they don't all contain the same set of information. All device descriptors have some basic information in common: the address of the device, which file manager to use with it, which device driver to use with it, which access modes are valid for the device and what it is named. Also, there is always a place for a table, called the initialization table, which contains information that the file manager and device driver might find useful.

The contents of the initialization table vary from one kind of device to another. Devices that use the Random Block File manager (e.g., Disk drives) have information like the device's stepping

rate and the number of sectors per track. The initialization table for devices that use the Sequential Character File manager contains a list of editing characters, Baud rate information, the number of lines on a page and other similar information. The PIPE device doesn't have anything in its initialization table except a byte indicating that it is a pipe.

HOW OS-9 USES THE DESCRIPTOR

When you first open a path to a device, OS-9 (specifically IOMAN) looks first at the device descriptor. It has to start there because all it has is the name of the device you want to use. The descriptor gives IOMAN the names of the file manager and device driver that it should use. IOMAN builds a path descriptor for the new path, copying much of the information contained in the device descriptor into the path descriptor, then hands the request off to the the appropriate File Manager with the address of the Device Descriptor and the Path Descriptor.

A file manager has access to all the information in the device descriptor, but it only uses the most generic part of it. There is no rule governing what the file manager can use, but generally it won't use data that is specific to the I/O device, such as its port. The file manager uses the device driver named in the device descriptor to do physical I/O operations. It passes the driver the address of the path descriptor, and (on the open call) the device descriptor. The driver reads values that might vary between the devices it is responsible for from the device descriptor and the path descriptor. Some of the values device drivers read from the device descriptor are the device address, Baud rate, stepping rate and port initialization byte.

MANAGING DEVICE DESCRIPTORS

The device descriptors that come with your system should be adequate to describe your hardware. If the company that sold you the system is doing its job, you will find that your copy of OS-9 has more device descriptors than you need. You may also find a directory of alternate descriptors somewhere on your distribution disk.

If the system comes with too many device descriptors in the system boot, you may want to remove the extra ones. The most common problem here is that OS-9 often comes with device descriptors for disks /D0 through /D3. Few people have more than two drives (/D0 and /D1), so the space used for the /D2 and /D3 descriptors is usually wasted. If you need every last byte of memory, remove unneeded descriptors by building a new bootstrap without them. If you save the modules on disk you will be able to load them later when you discover that you need them.

Unless a device is hardly used at all, it is best to include its descriptor in the boot file. If you include a module in the boot file, you can be certain that it will be packed into memory as efficiently

as possible and won't disappear if you unlink it by mistake. Also, the error messages you get when you try to use a device whose descriptor isn't in memory are sometimes hard to understand. I always seem to have trouble with missing descriptors when I am four or five hours overdue for bed. At times like that I only understand the simplest error messages. Sometimes I have gotten myself into a bit of a panic before I realized that the device descriptor was sitting safe on disk.

MAKING AND MODIFYING DESCRIPTORS

It is sometimes a matter of judgment whether to generate a new device descriptor or use one that you already have. If you have a new serial card, disk controller or whatever, you will definitely have to make a device descriptor for it. The device address isn't something you can change after a file has been opened. If you just want to turn on XOn/XOff support for your terminal, you have the option of modifying that attribute on the fly.

Information in the initialization table can be changed with the I$SetStt service request. The change isn't actually made to the device descriptor, just to the path descriptor's copy of the initialization table. If you aren't certain you want to make the change a permanent one, this is the way to do it.

If you choose to go the I$SetStt route, you still have a few choices. You can do it all by yourself with a piece of code like:

```
***************************************************************

lda #0                  Standard input path number
ldb #SS.Opt             Select the <Read option> getstat
leax OptBuff,U          Point at 32-byte buffer
OS9 I$GetStt            issue SVC
bcs IOError             If error; deal with it

***************

*  Just by way of example let's turn on XOn/XOff

*
ldb #$11                XOn
stb PD.XOn-PD.OPT,X     Save it in XOn slot of Opt area
ldb #$13                XOff
stb PD.XOff-PD.OPT,X    Save it in XOff slot of Opt area
ldb #SS.Opt             Select the <Write option> setstat

***************

*  A and X are still set from the previous call

*
```

```
OS9 I$SetStt          issue the SVC
bcs IOError

***************

*  All set

*******************************************************************
```

If you mean to change the characteristics of a device in the middle of a program, that is surely the way to do it; but, since the change is only to the path descriptor, the change will go away when the path closes. Even this isn't as simple as it seems. The standard I/O paths are all "dups." The I$Dup call is used to give the same path lots of path numbers. All three standard I/O paths often use the same path descriptor. Since standard I/O paths are inherited when a process is forked, the change made in this program is passed to the SHELL (or whatever program forked this one) and perhaps back through several generations. Clearly, path descriptors should be changed cautiously.

Still, the path will eventually be closed, and when that happens your change will go away. If that's fine with you, there is an easy OS-9 command, TMODE, which does just this kind of setstat for you. You can turn on XOn/XOff from the SHELL before starting a program and not have to worry about writing the SetStat into it. I assume that you have used TMODE by now; if you haven't, do it soon. There are some values that can be manipulated without causing great trouble. Try turning ECHO off:

OS9:TMODE -echo

When you are convinced that its no fun typing without seeing the results turn ECHO back on:

OS9:TMODE echo

It's also useful to experiment with PAUSE.

XMODE

If you want to make the change a little more permanent, you have to change the device descriptor. Some versions of OS-9 include the XMODE command. This command makes changes to the initialization table in the device descriptor instead of the copy in the path descriptor. Also, unlike TMODE, it can be used to change the attributes of a device that isn't one of the shell's standard paths. The syntax of XMODE looks a lot like TMODE, but the difference between the control blocks they act on means that TMODE takes effect immediately, but may not have a permanent effect. XMODE only takes effect when a new path to that device is opened, but will continue in effect for every path opened to that device until OS-9 is rebooted, or something else is done to alter the device descriptor.

If you don't have XMODE you can still change device descriptors on the fly. DEBUG changes device descriptors as easily as it changes any other type of module (though it would be hard to change path descriptors with it). DEBUG can change anything about the descriptor, including the device address and the access mode — values that can't be altered by XMODE.

DEBUG

You've got to keep your wits about you when you use DEBUG. There is nothing to protect you from yourself. If you feel any doubt about it, plan out what you will do before you start. The following is a script for changing the device address for the device descriptor "/T2":

```
OS9:debug
DB: l t2                 link to /T2
     EA74 87             first byte in t2 module (sync byte)

DB: . .+e                move forward to device address

     EA82 0F

DB: m . .+3              take a peek
EA80 3103 0FE0 241A 0000 0100 0101 0001 1808 1. . .$. . . . . . . . . .

DB: <CR>                 just a carriage return
     EA83 E0             the second byte of the device address

DB: <CR>                 another <CR>
     EA84 24             the last byte of the device address

DB: =34                  change that byte
     EA85 1A             the number of bytes in init table

DB: -                    check the byte we changed
     EA84 34             OK

DB: q
OS9:
```

Note: <CR>means the carriage return or enter key.

That is sufficient if you don't want to save the descriptor for another session. If you do want to save this modified version of /T2, its CRC bytes will have to be updated. DEBUG changed a byte in the module, and when it is next loaded OS-9 will reject it because the CRC will indicate that the module is flawed. The VERIFY command will fix the CRC:

```
OS9: save temp t2
OS9: verify u <temp >new.t2
```

One problem still remains. There are now two /T2 modules. One in the boot file, the other in the file "new.t2." If we leave things just the way they are there won't be any way to use the new /T2.

The version of /T2 in the boot can't be removed from memory by unlinking it. All modules in the boot are protected from that. It could be replaced by a module with a higher revision number, but the new version of /T2 we made has the same revision number.

There are two approachs we can take. If we really meant to change the address of /T2, the /T2 module in the boot file will have to be replaced. This can be done with OS9GEN.

The usual reason for changing the device address in a device descriptor is that there is a new port that needs a descriptor. In that case, what we really needed was a device descriptor with a new name as well as a new device address. DEBUG can be used to change the name as well as the address, provided that the new name is no longer than the old one. The following DEBUG statements could have been included at the end of the debug script for changing the address to change the module name from T2 to T5:

```
DB: . ea74            point at module start
    EA74 87
DB: . .+4             point at module name offset
    EA78 00           first byte of offset
DB: <CR>              to next byte
    EA79 2C           second byte of offset
DB: . .-5             back to module start
    EA74 87
DB: . .+2c            to module name offset
    EAA0 54           first byte in module name "T"
DB: <CR>              next byte
    EAA1 B2           a "2" with the high bit on
DB: =B5               reset to "5" with high bit
    EAA2 53
DB: -                 back up to check the change
    EAA1 B5           looks OK
```

If the module name needs lengthening this trick won't work. In fact, this method is altogether too cumbersome for most purposes. The only time I use it is when I want to experiment with a modified device descriptor without making any permanent changes. I put DEBUG in my startup file something like this:

DEBUG <my.mods >/nl

My.mods is a file containing the DEBUG commands to make the changes I have in mind. Make sure that the last command is "Q" for quit. DEBUG doesn't understand end-of-file. The redirected standard output goes to the null device that appears in the "Workshop" section of this book.

BUILDING A DESCRIPTOR FROM SCRATCH

The most powerful, and often the easiest, way to create a device descriptor is with the assembler. There are several examples of device descriptors in the Workshop section of this book. They form a good starting point for new descriptors. Take the descriptor that is closest to what you need, type it in, making whatever changes are necessary to fit it to your needs, and assemble it. You can test the new descriptor by loading it into memory with the LOAD command and doing some I/O to it. If it doesn't perform as you hoped it would, use UNLINK to remove it from memory and try again. Don't build a new boot file including the new device descriptor until you are certain it is correct; it's much harder to build a new boot than it is to use UNLINK and LOAD to replace a module that isn't part of the boot.

The values in the device descriptor are all important variables. They are covered several other places in this book, but perhaps it would do no harm to run through them all together.

THE CONTENTS OF A DEVICE DESCRIPTOR

The device descriptor starts like any other module — with a module header. The only special part of the module header is the type/language byte. The type is $F0, a type set aside for device descriptors. The language is $01 — 6809 object code — even though the module doesn't contain one executable byte.

Following the module header is the offset from the beginning of the module to the name of the file manager for this device (SCF, RBF, PIPEMAN, or IOPMAN). Next is the offset from the start of the module to the name of the device driver for this device (ACIA, CCIO, PIA, etc.). Then comes the mode byte for this device. This byte can have any of the values used as file access modes: read, $01; write, $02; execute, $04; public read, $08; public write, $10; public execute, $20; shareable, $40; directories, $80.

After the mode byte comes the initialization table. This table starts with a byte containing the length of that table. The table can be longer than 32 bytes, but any bytes past the thirty-second aren't copied to the path descriptor's option section. The first byte in every intialization table indicates the class of device the descriptor is for: SCF, RBF, PIPE, or SBF (sequential block file). The initialization table for SCF-type files contains all the bytes set by TMODE and XMODE. There is one value in the initialization table that can't be set by TMODE or XMODE, the offset to the "2nd device name string." The second device is the device used to echo input. Most terminal ports are set to echo to themselves. If you have a parallel keyboard and a graphics display, the keyboard probably echos to the display.

The initialization area for a RBF device contains information about the type of disk drive attached to it.

- The drive number indicates which of the drives attached to the disk controller this descriptor is for.

- The stepping rate of the drive is entered as a code. The relation between codes and stepping rates changes for different controllers, so check your OS-9 manual for details. In the case of the device driver distributed by Radio Shack for the Color Computer, this field is not used; the device driver only uses one stepping rate.

- The device type field contains three significant bits:
  bit 0 — 1 means the disk is eight inches
  0 means the disk is five inches
  There is no code for three inches
  bit 6 — 1 means standard OS-9 format
  0 means non-standard format
  bit 7 — 1 means it's a hard disk
  0 means it's a floppy

- If the disk is a floppy, the media density field indicates the recording density:
  bit 0 — 1 means double density
  0 means single density
  bit 1 — 1 means single track density (48 tpi)
  0 means double track density (96 tpi)

- The number of cylinders is the number of tracks recorded on one side of a disk. This number is usually 35, 40, or 80.

- The number of sides is one for single sided floppies, and two for double sided floppies. Hard disks can have many sides.

- Verification is usually used. If the disk controller or drive automatically verifies data it has written, verification need not be done; otherwise, it is strongly recommended. If you use good hardware you will seldom have data incorrectly written, but it is worth the check just to be sure. If the verify byte is zero, all writes will be verified.

- The number of sectors per track affects how much can be written on a disk. More sectors per track requires better hardware, and may be unreliable. Find out what the hardware manufacturer recommends.

- The number of sectors on track zero is special (except on the CoCo) because track zero is always written single density. By recording the first track at a standard density, regardless of the density of the rest of the disk, we make it easy for OS-9 to read enough of the disk to learn the characteristics of the rest of it. Since the density of track zero may be different from the rest of the tracks, the number of sectors on it may also differ.

- Sector interleaving is a trick to improve performance. If sector two is written directly after sector one on the disk there may be a serious performance penalty. After the computer reads the first sector in a file there is usually a tiny pause before it requests the second sector. If the pause is longer than the amount of time it takes the disk to cross the boundary between the first and second sectors, the disk will have to spin all the way around before the second sector can be read. By putting some other sectors (say the eighth and 15th) between the first and second, we give the program time to process the first sector, and ask for the second, before it is under the head and ready to read. The interleave factor specifies how many sectors apart sequentially numbered sectors should be located. The interleave factor doesn't change the actual numbering of the sectors on the disk. It just makes OS-9 map the numbers on the disk to the numbers it uses.

- The segment allocation size is part of another optimization trick. Most files start small and grow as more data is written to them. If the system is active with several processes writing to the disk, little pieces of files may get scattered around the disk. By making the segment allocation size greater than one, files can be made to start out with several sectors. If they are smaller than that, some disk space can be wasted. If most files are roughly some multiple of the segment allocation size, nice nonfragmented files are the result.

SUMMARY

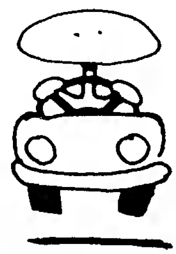
In this chapter you have learned a bunch of ways of making new device descriptors, and reasons for going to that trouble. You also heard about eliminating extra descriptors.

CHAPTER 23

adding a new device driver

In this chapter we discuss the reason for device drivers, then move on to reasons for creating new drivers. The actual business of writing a driver is left for the Workshop portion of this book.

WHY DEVICE DRIVERS?



Device drivers are operating system modules that deal with the actual hardware of I/O devices. Other parts of OS-9 deal with an idealized device. All SCF devices seem to perform the same operation in the same way from the point of view of every module except the device driver. The driver does whatever is necessary to make the real device look like the imaginary device that the rest of the OS-9 world sees.

This philosophy has some important implications. It gives OS-9 tremendous flexibility. Only one module has to be written to permit the system to use a new device. The only limit on the number of device drivers that OS-9 can support concurrently is the memory that they all take. Eventually, the drivers and their associated buffers and descriptors will use up more memory than you can tolerate. There is a hidden cost for this flexibility. When you know the characteristics of the device you are working with, there is a lot you can do to optimize your system. Isolating that knowledge in device drivers prevents the rest of the system from taking advantage of special features of a device.

On a system like the CoCo, positioning the cursor is a trivial operation. The screen is mapped into a block of memory; the cursor position is just an address. On a system with a terminal, positioning a cursor is a harder task. OS-9 doesn't concern itself with cursor positioning, not because it isn't important, but because a method general enough to work on systems with very "dumb" terminals would be fantastically wasteful on systems like the CoCo.

Cursor positioning is an example of a device characteristic that OS-9 hasn't taken responsibility for, but there are other things like buffering and error handling that OS-9 hides in the device driver at some cost in speed and power.

The actual design of a device driver will be taken up in the Workshop. It isn't difficult, but must be done carefully. OS-9 doesn't include good facilities for debugging parts of the operating system. Any bugs you write into the driver tend to have hair and TEETH!

WHY CREATE NEW DRIVERS?

If you like to play with your operating system, device drivers are a good playground. The operating system is meant to be expanded by having drivers added to it, and there are many opportunities for improvement. The ACIA driver that appears later is a version of the Microware version 1.2 standard that I modified to include <break> support. I don't know of any disk driver that includes a cache, but there isn't any reason they shouldn't. The size of the input and output buffers maintained by ACIA drivers is a subject that gets a lot of discussion.

If this kind of thing excites you, study the drivers presented here, and go improve your own. Two more warnings: Microware considers device drivers part of the operating system. They don't feel obligated to maintain compatibility with their old drivers, and especially not with your drivers. The other warning is also about compatibility: make a special effort to keep your driver compatible with Microware's distributed drivers. If you add enhancements that create a slight incompatibility, you may find that a program you buy relies on the feature you changed. Look at my ACIA driver for one way to add a feature without disturbing things too much.

Device drivers can be a particularly important part of a real-time control system. Not only do they sometimes have special devices to support, but device drivers are entered only a few cycles after an interrupt takes place. If you need to respond to an interrupt with some almost-instant action, the device driver is the only place to do the processing. Normal processes are run and put to sleep at the whim of the dispatcher; device drivers are run as soon as the source of the interrupt is discovered.

If, for example, you are controlling an outgoing voltage based

on an incoming voltage, the times required for A/D (Analog-to-Digital) and D/A (Digital-to-Analog) conversion may be almost more than you can afford. A device driver could be designed to drive both devices and perform some simple computations. It would pass information on to a normal program for low-priority processing, but would respond almost instantly to each interrupt.

If you create a new device driver with a new name (say Null), you must also build a device descriptor for it. No device can be used without a descriptor.

If you build or buy a new device driver, you will probably want to install it in your boot file. If you want to experiment first, or don't want to use space in the boot for a seldom used device, device drivers can be loaded after the system is booted. Just make sure that both the device driver and the device descriptor stay linked as long as they are being used. One good cause for inexplicable errors is that the driver has come unlinked and disappeared from memory. If this happens, unlink the descriptor and load and link both the descriptor and the driver again. Since modules in the bootstrap can't be dropped from memory no matter how many times they are unlinked, the problem doesn't show up there. If you load them separately, be careful.

SUMMARY

In this chapter we discussed the philosophy behind device drivers. I tried to show why you would want to write a new driver, and finished with some practical issues about driver design and installation.

CHAPTER 24

processes

In this chapter you will learn what processes are, how to create them and how to control them. There is also a discussion of system tuning, the art of getting the best possible performance out of your computer, and some information about signals.

WHAT IS A PROCESS?

Sometimes it seems that everything that isn't a module in OS-9 is a process. Processes do things. All programs involve at least one process while they are running. Frequently the words "process" and "program" are used interchangeably. All programs use CPU time; processes are things that use CPU time. The difference is that a program is a higher-level entity. A program might involve several processes, but it certainly should DO something. A process may sit around doing nothing most of the time. SYSGO is a process that sits around waiting for the world to fall apart so it can rescue you. OS-9 itself is an odd sort of process.

When you run a program from the SHELL:

OS9: list /d0/defs/os9defs

the shell "forks" a process to run the LIST program, then pauses until it completes. If you change the command:

OS9: list /d0/defs/os9defs >/p

you have some options. Just sit there and you will wait until the file has printed before you can go on. Hit control-C and the shell will

stop waiting. You get another prompt and can run another program (start another process). You can do this because the list program hadn't done any I/O to your terminal. If it had, the control-C would have acted as it usually does to abort the listing.

If you know in advance that you want the listing to run in "background," you can tell the shell not to pause after starting the process by putting an ampersand (&) after the command:

OS9: list /d0/defs/os9defs >/p&

You will get an "OS9:" prompt immediately, and the listing will proceed without your attention. If you decide that cutting the list process loose was a mistake, you can change your mind with the (w) shell directive. Just type 'w' on the command line:

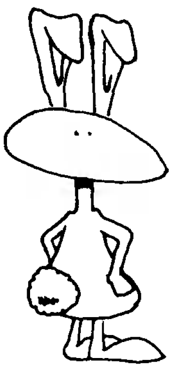
OS9: w

The shell will wait for a process it started (child process) to terminate before continuing.

FAMILY RELATIONSHIPS

Processes are related by the same names as a family. There is a parent process (which used to be called a father), and child processes (used to be sons). When a process forks a new child, it is said to have "spawned" a child. Two processes spawned by the same parent are said to be siblings (brothers).

THE INFAMOUS RABBIT PROGRAM



This might be a good place for an example. In the early days of mainframes, mischievous students would confound the computer operators by starting programs called "rabbit jobs." These programs would start at least two copies of themselves before going away. Modern systems have some protection against this kind of abuse, but OS-9 is helpless before it. There is no good purpose for rabbit jobs, but it may be the only way you'll find out how many processes your machine can run at one time.

Clean up your system before you run this. The Rabbit program is as hard to kill as a family of real rabbits. You may have to boot the system to stop them, but try KILL 0 first.

If you share a multi-user system, remember each of those rabbits will have your user number.

```
Rabbit - Rabbit Program to demonstrate F$Fork SVC

                         nam   Rabbit
                         ttl   Rabbit Program to demonstrate F$Fork SVC
                         IFP1
                         use   /d0/defs/OS9Defs
                         ENDC
```

```
  0011                Type     set    Prgrm+Objct
  0081                Revs     set    ReEnt+1
  0000 87CD003A                mod    ModLen,Name,Type,Revs,Entry,MemSize
  000D 52616262       Name     fcs    /Rabbit/
  0013 01             Version  fcb    1
       ****************
       *   Variable Memory
       *
D 0000                         rmb    200          Stack space
D 00C8                MemSize  equ    .
  0014                Entry
       ****************
       *  This program runs so fast that if it doesn't pause for
       *  a while the system instantly floods with jobs.  This sleep
       *  doesn't prevent the flood, but it does slow it enough that
       *  you can see what's hitting you.
       *
  0014 8E0190                  ldx    #400
  0017 103F0A                  OS9    F$Sleep
       ****************
       *  Prepare to fork a rabbit
       *
  001A 8611                    lda    #Type        Type for forked module
  001C C600                    ldb    #0           Memory overide for forked modu
  001E 308DFFEB                leax   Name,PCR     name of module to fork
  0022 108E0000                ldy    #0           length of parameter area
       ********
       * Since there is no parameter area don't bother with a pointer
       * to it.
       *
  0026 103F03                  OS9    F$Fork
  0029 2509                    bcs    Exit         If error; don't fork again
       ********
       *  A and X have been changed by the F$Fork call
       *
  002B 308DFFDE                leax   Name,PCR
  002F 8611                    lda    #Type
  0031 103F03                  OS9    F$Fork
  0034                Exit
  0034 103F06                  OS9    F$Exit
  0037 E10CE4                  EMOD
  003A                ModLen   equ    *

00000 error(s)
00000 warning(s)
$003A 00058 program bytes generated
$00C8 00200 data bytes allocated
$21E5 08677 bytes used for symbols
```

Use the PROCS command to watch the number of rabbits running around in your system grow. If you want an exercise in frustration, let them get started, then try to kill them off with the kill command. When you give up, boot your system. If you want an interesting challenge, write a rabbit-killer program. It can be done, though you may need to cheat. If you struggle with this program and find that the rabbits get ahead of you, try increasing your priority. If that doesn't work, disable interrupts and use a shotgun approach.

CHAINING NEW PROCESSES

The shell starts programs by forking them. It can also be made to chain to a new program. The F$Chain service request eliminates the calling process and replaces it with the process requested by F$Chain. If you do this with the SHELL, you won't have a SHELL to return to. With other programs it is a way to save memory. F$Fork followed by F$Exit would act almost like F$Chain except that, between the Fork and the Exit, memory would be allocated for both the parent and the child process. Because of this, F$Chain is sometimes the only way to go when memory is very tight. There is also a small difference in the time required for the Fork and Chain requests. Chain is a little faster, but not enough to make a difference.

ALLOCATION OF PROCESSOR TIME

There are plenty of ways for you to start several processes running at once (Rabbit is an extreme case.). One of OS-9's primary jobs is to divide system resources among however many processes you choose to run. OS-9 distributes memory and I/O devices on a first come, first served basis. Time on the microprocessor is distributed according to a combination of "demand" and time-sliced rules. Some things can't be delayed: keyboard input, for instance, has to be read before another character is typed. OS-9, itself, provides time-critical services. It is able to interrupt any process and take whatever time it needs.

All processes other than OS-9 have to wait for a "time slice." OS-9 requires a system clock. It doesn't necessarily have to keep track of time-of-day or calendar information; it does have to generate an interrupt at least every 10th of a second. OS-9 uses these clock interrupts to tell it when to put the current process to sleep and start the next.

Processes wait in a queue (a line) for a chance at the CPU. The queue is arranged according to the "age" of the processes; that is, how long they have been in the queue. The processes that have been in the queue longest are at the front. There is, however, an exception to this scheme.

When a process finishes its turn, it doesn't necessarily go to the end of the line. Each process enters the queue with an initial age. If a process's initial age is greater than the age of some process that has been in line for awhile, it gets to "cut in line." This may seem unfair to you. If it upsets you too badly, don't use the feature. The initial age of a process is its priority. You can set the priority of a process with the SETPR command. Raising the priority of a process just a little with SETPR is usually enough to make it run substantially faster.

Careful adjustment of process priorities can increase the efficiency of your work. OS-9 doesn't know what a process is supposed to do, or how important it is to you. Without instructions it will treat them all the same.

Some processes don't need frequent access to the processor. A process that is running your printer is a good candidate for a low priority. You may think that 160 characters per second is fast; the computer doesn't. A process that only has to send characters to the printer at that speed will spend most of its time waiting. Even if it isn't right on the spot with a character to be written, it's no big deal if the printer has to wait a fraction of a second. Set the priority for this type of process very low. It will get a crack at the processor every now and then. During its turn, the process will feed the device driver for the printer a bufferful of characters, which the driver will print until the process's next turn.

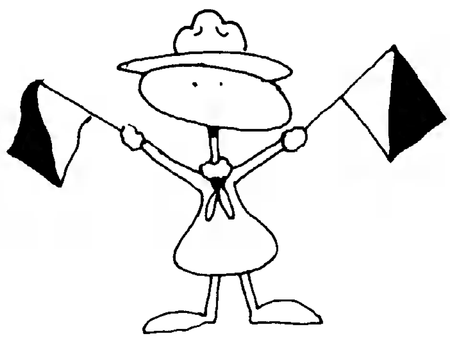
A process that you (the user) interact with needs high priority. It probably won't use most of its turns because it will be waiting for input from you. (Actually, it won't even get a turn if it's waiting for input.) When you do give a process something to do, you want it done fast. Editors usually sit around waiting for a keystroke, but it would be annoying if a key took half a second to register, or the screen updates took a few seconds. People need attention! Make sure your computer knows this by assigning a high priority to any programs you will be working with.

Programs like COPY, assemblers, and particularly the C compiler will take all the CPU cycles they can get. If you want to run them together with something else, you will need to protect the other processes by giving the CPU-intensive process a lower priority than the others.

As a rule, programs that perform a lot of computations need to be interrupted and put at the back of the queue. Processes that do a lot of I/O, particularly SCF style I/O, are already slowed by the device they are driving. They don't need to be controlled artificially.

Process priority has several uses. It can tame CPU hogs. It can take a task that you aren't impatient about and tuck it down where it will get to the CPU so infrequently that you won't notice its impact on the system. If you want close attention paid to something you're working with, such as a text editor or real-time graphics, a high priority will give it lots of attention.

You won't speed anything up by assigning all your processes high priority. As the word priority implies, you have to rank processes. If you assign all processes a priority of 250 they will run just the same as if you gave them all 50.



Processes are isolated from one another. Part of the concept of processes is that each should appear to be running in its own computer. There is a carefully constructed leak in this isolation. Signals can be sent between processes. OS-9 sends signals: keyboard abort, and keyboard interrupt. There is also a system abort signal that never gets to a process. It kills the process without warning. Keyboard abort and keyboard interrupt will kill an unprepared process, but a signal trap can be set up to catch these signals and do whatever you like with them (including ignoring them).

Signals can be sent by any process to any other process. The only restriction is that only a process running under user number zero can send system abort signals to processes running under any user number. You can kill your own processes, but unless you have special privileges you can't kill processes belonging to other users.

Signals are the peep-hole that OS-9 leaves between processes. It is such a narrow communication channel that it takes ingenuity to use, but it is enough to build powerful systems of processes. The building blocks are the F$Send system service request, which will send a specified signal to a specified process; the F$Icpt request, which sets up a signal-intercept trap; and F$Sleep, which causes a process to wait for a signal.

Without an intercept trap, a process will be killed by the first signal it receives. A trap doesn't have to be very complicated. A simple rti (return from interrupt) instruction is sufficient to prevent the process from being killed by any signal it receives. It isn't necessary to put a process to sleep for it to receive signals, but frequently processes will do everything they can find to do, and then they'll need to wait for a signal from another process before continuing. A sleeping process doesn't use any CPU time, and it responds as quickly as possible to signals.

The Workshop section of this book contains several examples of programs that use signals. Even the stripped-down programs there are too big to stick in the middle of a chapter.

Merely ignoring signals isn't so hard. The following program segment sets up a trivial signal-intercept trap that saves the signal code, but doesn't do anything about it. Other parts of the program check the "Signal" byte from time to time and do something appropriate.

SigTrap intercepts signals and that's all. It sets up a trap, then waits until a signal arrives. When it receives a signal, SigTrap writes a brief message explaining the signal and quits.

The U register only needs to be set before calling F$Icpt under

exceptional circumstances. It is so painful to have different offsets from U in the main program and the trap, that the small performance improvements that might come from adjusting U (smaller offsets from U in the trap) are generally ignored. The only programs that adjust U are those that don't use U as the base register for global storage (C programs).

Sleeping with X set to zero means sleep forever. A signal will wake a sleeping process, so this actually means sleep until a signal arrives.

The F$PErr service request prints an error message based on the number in B. This service request formats the error number. Including code to format decimal numbers in SigTrap would more than double the size of the program.

```
                                nam   SigTrap
                                ttl   Display signals
                                IFP1
                                use   /d0/defs/os9defs
                                ENDC
   0011               type      set   Prgrm+Objct
   0081               Revs      set   ReEnt+1
   0000 87CD00A8                mod   ModLen,Name,Type,Revs,Entry,MemSize
   000D 53696754      Name      fcs   /SigTrap/
   0014 01            Version   fcb   1
   0015 0A            Intl      fcb   $0A
   0016 57616B65                fcc   /Wake up/  we won't see this one
   001D 0D                      fcb   $0D
   001E 0A            Int2      fcb   $0A
   001F 4B657962                fcc   /Keyboard Abort/
   002D 0D                      fcb   $0D
   002E 0A            Int3      fcb   $0A
   002F 4B657962                fcc   /Keyboard interrupt/
   0041 0D                      fcb   $0D
   0042 0A            Intx      fcb   $0A
   0043 4D697363                fcc   /Misc. Signal/
   004F 0D                      fcb   $0D

 D 0000               SigCode   rmb   1
 D 0001                         rmb   200        stack
 D 00C9               MemSize   equ   .

   0050               Entry
   0050 308D004E                leax  Trap,PCR
   0054 103F09                  OS9   F$Icpt
   0057 2546                    bcs   Error
   0059 308DFFB0                leax  Name,PCR
   005D 108E0007                ldy   #7
   0061 8601                    lda   #1         std out
   0063 103F8A                  OS9   I$Write    Token write
   0066 252D                    bcs   End
   0068 8E0000                  ldx   #0
   006B 103F0A                  OS9   F$Sleep    wait for a signal
   006E D600                    ldb   SigCode
   0070 C101                    cmpb  #1
   0072 2606                    bne   S2
   0074 308DFF9D                leax  Intl,PCR
   0078 201B                    bra   End
   007A C102          S2        cmpb  #2
```

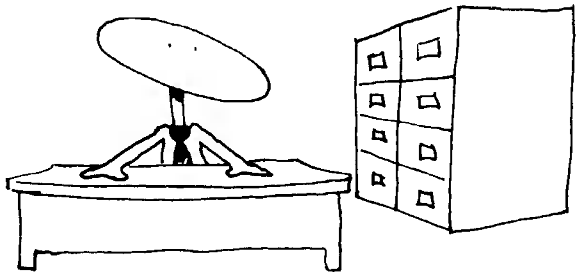
```
007C 2606                bne    S3
007E 308DFF9C            leax   Int2,PCR
0082 2011                bra    End
0084 C103       S3       cmpb   #3
0086 2606                bne    Sx
0088 308DFFA2            leax   Int3,PCR
008C 2007                bra    End
008E 308DFFB0   Sx       leax   Intx,PCR
0092 103F0F              OS9    F$PErr
0095            End
0095 108E0050            ldy    #80       Max length
0099 8601                lda    #1        std out
009B 103F8C              OS9    I$WritLn
009E 5F                  clrb             clear carry
009F            Error
009F 103F06              OS9    F$Exit
00A2            Trap
00A2 E7C4                stb    SigCode,U
00A4 3B                  rti
00A5 AC50ED              EMOD
00A8            ModLen   equ    *

00000 error(s)
00000 warning(s)
$00A8 00168 program bytes generated
$00C9 00201 data bytes allocated
$227B 08827 bytes used for symbols
```

SUMMARY

In this chapter you learned about processes. They are the active objects in a computer. They are started with F$Fork, or F$Chain. Adjustment of process priorities can pay off when several processes are running. Processes are carefully isolated from one another, but they can communicate with signals.

CHAPTER 25

file managers



This chapter is an overview of file managers. Each of the main file managers is mentioned, and the role of a file manager in the OS-9 system is discussed.

WHAT'S A FILE MANAGER?

File managers are the level between the I/O manager and device drivers. Like device drivers, they hide some aspects of the I/O system from user programs. Some requests are passed on to device drivers with little intervention: SCF (Sequential Character File) I$Write requests would be an example. Other requests, such as the Delete request to the RBF (Random Block File) manager, are handled mostly in the file manager, with only incidental requests going to the driver. The I$Seek request and some GetStat and PutStat calls don't go to the driver at all.

Much of SCFMan's function relates to editing. All special characters, like backspace and reprint-line, are handled here. SCFMan also handles contention between several processes wanting simultaneous access to a device.

RBFMan is the only part of OS-9 that knows anything about the structure of a disk. It handles directories, file descriptors and the disk identification sector.

Pipeman does everything for pipes; the device driver for pipe files does exactly nothing. The device driver is necessary because

IOMan wouldn't tolerate a path without one, but there isn't any actual device associated with a pipe. Pipes are manufactured entirely of mirrors.

Most devices can be fit into the SCF or RBF class. Devices in these classes can be added to an OS-9 system with little effort. At most, a device driver will need to be written, and very likely, only a device descriptor.

A few devices can benefit from an entirely new file manager. OS-9's architecture permits new file managers to be added without any disruption. Several ideas for alternative file managers come to mind.

POSSIBILITIES FOR NEW FILE MANAGERS

A special file manager that has already been written is the I/O processor file manager. Many of the functions of a file manager can be pushed all the way down into a device if the device is intelligent enough. Intelligent controller boards that support terminals and printers have been made. The processor on the board can handle line editing nicely without any help from the file manager. When this function can be placed in the controller board, it increases the I/O capabilities of the computer substantially — if the file manager is stripped down to just those funtions that can't be done in the device controller. Each CPU cycle that can be made the responsibility of the I/O processor becomes another cycle available for user programs. Taking full advantage of an intelligent SCF device requires a special file manager with line-editing functions removed and possibly some additions to give the controller the information it needs.

The Random Block file manager can benefit just as much from intelligent controllers as the SCF manager, perhaps more. At the least, conversion of logical sector numbers to physical disk addresses can be done in the controller. Other functions like error recovery, file and directory handling — in fact, most of the functions of the RBF manager — could be moved to the controller. A dedicated processor could handle these operations more efficiently than the general purpose processor running the file manager. Even if the microprocessor on the controller is slower than the main processor, unloading functions onto intelligent peripherals returns processor resources to other programs.

It is in the interest of the manufacturers of intelligent controller boards to write special file managers, as well as device descriptors, for their hardware. The special software makes their hardware look a lot better.

Other file managers would add new features to OS-9. A suitable file manager would add local area network support to OS-9. This would permit several computers to be attached to each other. Resources like disk space and peripherals could be shared

through the network. A single, large capacity disk could serve several computers, saving money on disks and making public files available to users of any computer on the network. Other high-priced peripherals, like fast printers and graphics devices, would also be easier to afford if they could be shared by several computers on a network.

It wouldn't be wise to make a network depend on some particular hardware. So long as standard device drivers are used, any device that supports the set of functions required by the network file manager could be used. At this moment (winter of 1985), a network file manager for OS-9 isn't available, but when one is written it should be possible to add it to any OS-9 system with the required hardware.

A more generalized version of the pipe would be a useful addition to OS-9. Features like the ability to communicate with any process by number would add significantly to the usefulness of pipes.

WRITING A FILE MANAGER

There is nothing inherently difficult about writing a file manager. The trickiest aspect to the job is that the debugger doesn't work for system modules.

A file manager has 13 entry points, each branching to a routine that provides a specific service. All the entry points must be there, but the attached routines can be null procedures that just return with carry clear, or routines that set an error code and return with carry set.

If a new device driver will do what you want, don't write a file manager. A file manager is a much bigger project than a driver; just look at the relative sizes of the file managers and drivers included with your system.

A new file manager is about the most important addition that a user can make to OS-9. A new or modified file manager can add new functions to OS-9. A new file manager can be brought into use by adding a device descriptor that references it. Clearly, this is one of the directions in which OS-9 was meant to be expanded.

SUMMARY

In this chapter you learned what role file managers play in the OS-9 environment.

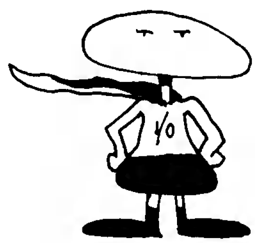
CHAPTER 26

the i/o manager

In this chapter we discuss the role of the I/O manager in an OS-9 system. The philosophical role of IOMan and many of its practical duties are covered.

WHY IS THERE AN IOMAN?

The Input/Output Manager, IOMan, does just the things you would think a manager should do. It is called as part of system startup, and makes itself responsible for all the I/O system calls. As each call to an I/O system service is made, IOMan catches it, collects the necessary resources and hands it off to the appropriate file manager.



IOMan sits right at the top of the I/O hierarchy. It processes every I/O system service request and routes all I/O interrupts. For most of them it only executes a handful of instructions before passing off to a file manager. Only for the Attach, Dup, Detach, Open and Close requests does it do any substantial work.

ATTACH/DETACH

The Attach request is an oddball. It is almost never used except by the I/O manager. It places a new device in the device table and calls the device driver to initialize it. The device table is a quick reference table used by the I/O manager to determine whether a device has already been attached. Knowing whether a device has been attached prevents IOMan from wasting time at-

taching it again. More importantly, it prevents IOMan from having the driver reinitialize the device. All the facts that Attach collects about a new device will be contained in the device table; these facts can then be taken from the table whenever the device is opened in the future.

Detach is the inverse operation for Attach. It removes a device from the device table and calls the termination routine for the device.

Users seldom attach a device; however, they often open files. When a file is opened, IOMan attaches the device, if necessary. It also creates a path descriptor for the new path and calls the file manager to do anything it might need to do about a new file, e.g., find the location of a RBF file on disk.

DUPING PATHS

When a path number needs to be changed, the Dup call is used. IOMan is responsible for Dup. It assigns an additional path number to a path descriptor and increments the use count of the path descriptor. This sounds like a trivial operation, but it is the only way to save the information about a path if you have to close it. Say you want to open a path to the printer as standard output without losing the current standard output file. This can be done by:

dup path 1 <standard output>,
save the new path number <x>
close path 1
open the printer

The printer will appear as path 1 because IOMan always assigns the lowest available path number, and path 0 is already taken by standard input.

When you want to restore the original standard output file:

close path 1
dup path <x>

Path <x> is the dup of the original standard output. It will be duped to the lowest available path number. Since we just closed path one, releasing that path number, this dup will be to path 1.

close path <x>
... and that does it.

Close is another operation performed by the I/O manager. If the use count of the path descriptor for the path being closed is greater than one, the I/O manager just decreases it by one. If the use count is one, there aren't any other paths using the descriptor, so IOMan returns the memory for the descriptor. If there are no other paths using the device, IOMan will also detach the device.

The real working I/O operations are passed right through the I/O manager to the file manager. IOMan wouldn't know a directory if it bit "him" on the nose, and it passes on read and write requests as fast as possible. Its only involvement with the bulk of commands is to get the address of the path descriptor and pass it along to a file manager.

The I/O Manager takes requests for services, arranges the paper work and gets the right team of modules together. Its main direct involvement is at the start and end of a project (Open and Close). Sounds like the word "manager" was correctly included in the name, doesn't it?

SUMMARY

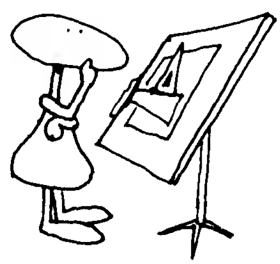
In this chapter you discovered that IOMan is a true executive-type module. It seldom does anything but supervisory work, but its organizational abilities hold the OS-9 I/O system together.

CHAPTER 27

disk formats

In this chapter we discuss the way OS-9 structures data on disks. We take a low-level look at file allocation and directory structures.

PHYSICAL DISK FORMAT



Formatting a disk changes it from a piece of useless junk into a carefully organized empty file structure. There are some features of formatted disks that are common across all hardware and operating systems.

A disk's surface is divided into tracks, which are concentric circular paths around the disk reminiscent of the grooves in a phonograph record. The number of tracks depends on the quality of the disk drive. The more tracks the more data can be stored on the disk, and the more precision the disk drive must have to position the head over a track.

Each track is divided into sectors. These sections of the track are the pigeon holes where data is stored. The data part of each sector is surrounded by timing and identifying bytes. These bytes help the disk controller find a sector it is searching for, leaving some room for error. All OS-9 sectors are 256 bytes long, but the number of sectors on a track varies widely depending on the size of the disk and the recording density that is used. The smallest number of sectors per track currently being used is 16 for single density 5¼-inch disks. Eight-inch double density disks have 28 sectors per track.

THE IDENTIFICATION SECTOR

After we get above the level of the physical disk all OS-9 disks have the same characteristics. Information about the disk as a whole is stored in its first sector, that is, the first sector on the first track. The sector containing this information is called the "identification sector." The information in the identification sector includes the specifications for the way the rest of the disk is written, the location and size of the bootstrap (if it's there), the name and creation date of the disk, the user number of the owner of the disk and a pointer to the root directory.

One of the fields in the identification sector is DD.BIT. This field indicates the number of sectors in a cluster. For most systems this field will be one, but if your disk is exceptionally large, this value can be made greater than one. Clusters of sectors are treated like sectors for many purposes. In particular, when sectors are allocated to a file, they are allocated a cluster at a time. If a disk were so large that OS-9 couldn't keep track of all the sectors on it, clusters could be formed of two sectors each; this would double the size of the disk that could be handled. The number of sectors per cluster can be set to any value necessary to permit OS-9 to handle the disk.

THE ALLOCATION MAP

Past versions of OS-9 could handle a maximum of only 2,048 clusters. With so few clusters available, even a double-sided, double-density 8-inch disk needed to have two sectors per cluster. More recent versions of OS-9 have increased the maximum number of clusters to 524,288 (128 megabytes at one sector per cluster). This enhancement was made by increasing the number of sectors dedicated to the allocation map from one to a maximum of 256.

The sector right after the identification sector contains the beginning of the disk allocation map. This is an array of bits that indicates whether each cluster on the disk is allocated or free. If the bit corresponding to a cluster is one, the cluster is allocated; if it is zero, the cluster is free.

The disk allocation map is used whenever a file is created or deleted, or when the size of a file is changed. In each of these operations disk space is used or freed. The disk allocation map contains the location of each free cluster on the disk.

THE ROOT DIRECTORY

One of the fields in the Identification sector is a pointer to the root directory. Every disk, even one on which you never create a directory, has a root directory. If you do a directory command on /D0, the result will be a list of the files in the root directory for /D0. This directory is called the root directory because if you view the

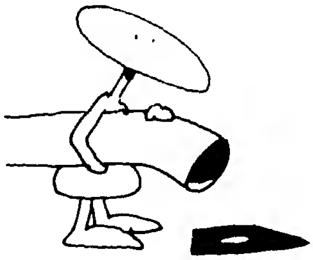
directories on a disk as forming a tree, the root directory is at the base (or root) of the tree.

Outside its special position there is nothing exceptional about a root directory. It is a file of directory entries, each entry consisting of a 29-byte file name and the three-byte logical sector number of the file descriptor for the file. Usually, a directory contains many empty directory entries. The empty entries are distinguished by the $00 in the first byte of the file name.

A directory file has a special purpose, but it can be read or even written much like any other file. Directory files should only be written to with great caution, but reading them is harmless. A directory file can be examined with the following simple program:

```
                                nam    DirDump
                                ttl    Dump the working directory to standard ou
                                IFP1
                                ENDC
     0011             type      set    PRGRM+OBJCT
     0081             Revs      set    REENT+1
     0000 87CD0045              mod    MEnd,Name,Type,Revs,Entry,Memsize

   D 0000             DPathNo   rmb    1          Directory path number
   D 0001             Buffer    rmb    32         buffer for directory entries
   D 0021             Stack     rmb    150
   D 00B7             Memsize   equ    .

     000D 44697244    Name      fcs    /DirDump/
     0014 01          Version   fcb    1
     0015 2EA0        Dirname   fcs    /. /

     0017             Entry
          **********
          *  Open working directory file
          *  for reading
          *
     0017 308DFFFA              leax   Dirname,PCR
     001B 8681                  lda    #DIR.+READ.
     001D 103F84                OS9    I$Open
     0020 251D                  bcs    Error
     0022 9700                  sta    DPathNo    Directory file path number
          *********
          *  Set up for copy loop
          *
     0024 108E0020              ldy    #32        length to read and write
     0028 3041                  leax   Buffer,U
     002A             CpyLoop
     002A 9600                  lda    DPathNo
     002C 103F89                OS9    I$Read
     002F 2509                  bcs    TestEof    error; test for EOF
     0031 8601                  lda    #1         Std output
     0033 103F8A                OS9    I$Write    write a directory entry to Std
     0036 24F2                  bcc    CpyLoop    no error; copy next entry
     0038 2005                  bra    Error      error; abort
     003A             TestEof
     003A C1D3                  cmpb   #E$EOF     if the error isn't EOF
     003C 2601                  bne    Error      it's a real error
     003E 5F                    clrb              otherwise it's not an error so
     003F             Error
     003F 103F06                OS9    F$Exit     return
     0042 7EC002                EMOD
     0045             MEnd      equ    *
```

```
00000 error(s)
00000 warning(s)
$0045 00069 program bytes generated
$00B7 00183 data bytes allocated
$223F 08767 bytes used for symbols
```



Dirdump is best used with a pipe to DUMP:

OS9: DirDump ! DUMP

The result will be a dump-format listing of the working directory. DirDump can be used without a pipe to DUMP, but the non-printable characters in the directory may well drive your terminal berserk.

THE FILE DESCRIPTOR

Directory entries don't point directly at files. They point at file descriptor sectors which give all the information about a file except its name and the directory it's in. The most interesting result of keeping most of the information about a file out of the directory is that a file can be renamed, moved about and even given aliases without special effort.

Think about what would happen if two directories had entries pointing to the same file descriptor. That file could be accessed under two names from two separate directories. This kind of trickery upsets some OS-9 commands, notably DCHECK, but in most cases OS-9 handles it smoothly. There is even a field in the file descriptor that can be used to give the number of directory entries pointing to it. A file's space allocation won't be returned until that counter is zero.

SUMMARY

In this chapter you learned about the physical organization of a disk and the logical organization OS-9 imposes on it. You read about the data in the disk identification sector, the allocation map, file descriptors and directory files.

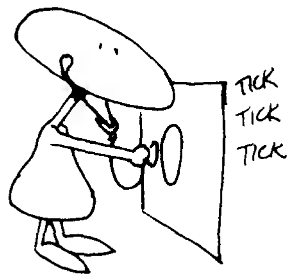
CHAPTER 28

interrupts

In this chapter you will learn what interrupts are and what they have to do with I/O and multitasking.

INTERRUPTS

Interrupts are what make OS-9 tick. As you will discover soon that is something of a pun.



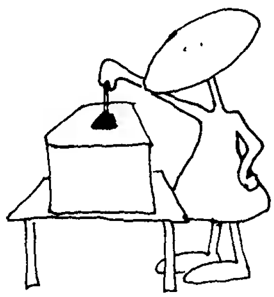
The 6809 microprocessor can deal with three different types of hardware interrupt: IRQ, interrupt request; FIRQ, fast interrupt request; and NMI, non-maskable interrupt. Frequently, the hardware OS-9 is running on can produce all three types of interrupts. There are versions of OS-9 that make some small use of FIRQ and NMI interrupts, but the IRQ interrupt is the primary hardware interrupt in OS-9 systems.

All interrupts (including software interrupts) do about the same thing. They push the MPU registers, fetch an address from a special location in memory and jump to that address. Each type of interrupt gets the address it jumps to from a different location. This makes it easy to handle different types of interrupt with different routines.

The FIRQ interrupt is a little special. It is meant to be used for devices that need exceptionally fast service. FIRQ just pushes the program counter (PC) and condition code (CC) registers. Other interrupts push all the registers. Saving fewer registers makes servicing a FIRQ request faster than the other interrupts.

The hardware interrupts enter the microprocessor on special lines. There is a pin on the 6809 chip for each of the three hardware interrupts (plus one for reset, which acts a little like an interrupt). The 6809 instruction set includes three instructions that cause software interrupts (SWI, SWI2, and SWI3). These instructions push the registers and jump through a vector just like hardware interrupts. The OS9 instruction used in assembly language programs is actually shorthand for a SWI2 instruction followed by a byte with the SVC number.

POLLING



Some operating systems (e.g. CP/M) use a technique called polling. In this type of system each piece of hardware that might need attention is polled (or checked) at frequent intervals. In a system with a printer on a parallel port and a terminal on a serial port the overhead involved in polling the I/O devices isn't too much. The serial port will need to be checked a few thousand times per second; the printer (being a slow output device) can be polled much less frequently — 200 times per second should be enough. When a program is running, it must take responsibility for any polling. There is no automatic way for the operating system to take over at intervals.

THE ALTERNATIVE

Under OS-9, interrupts are precisely a way for the operating system to take over when it is required. I/O ports are programmed to produce an IRQ interrupt every time they need attention. When the IRQ is received, OS-9 is given control of the machine so it can handle the event. Since an interrupt is only generated when something actually needs doing, there is no constant need to watch for devices that need service. Neither user programs nor OS-9 need to do any polling.

The most important result of the use of interrupts is that I/O is simplified for users. OS-9 deals with all I/O hardware. If a program isn't ready for input when a byte arrives, OS-9 holds it in a buffer until the program requests it. Similarly, OS-9 will maintain a buffer of characters ready for output if a program is producing output faster than the output device can take it.

MULTITASKING AND THE CLOCK

OS-9 multitasking capability is also based on interrupts. I/O hardware may produce frequent interrupts in an OS-9 system, but they can't be relied on. Minutes might pass without a single I/O operation. To ensure that OS-9 gets control at frequent intervals, every OS-9 system has a special device that produces interrupts. These interrupts, called timer interrupts, come 10 times per second on a Level One system, and 100 times per second under Level Two. OS-9 uses the timer interrupt as a trigger for its housekeeping operations. When several processes are running, OS-9

switches the current process off and starts another every few clock ticks (different numbers of ticks for different systems).

Switching from one process to another several times per second makes it possible for OS-9 to appear to run several programs at the same time. If you were a very fast-moving house painter, you could work on all four walls, moving quickly around the house. If you moved fast enough, a spectator might think that four painters were working slowly, one on each wall.

If you are thinking about the huge amount of time that painter would waste running from wall to wall, you are right. OS-9 uses important time servicing timer interrupts and switching context from one process to another. The more frequent interrupts in Level Two systems are one of the reasons programs run a little faster under Level One. Still, the problem isn't that bad. A 10th of a second is a long time for a computer. Executing a hundred or so instructions to start a new process every 10th of a second isn't a heavy burden for a processor that executes hundreds of thousands of instructions per second. In summary, supporting multitasking has some cost, but it isn't a large price to pay.

THE POLLING TABLE

When an IRQ interrupt comes in, OS-9 only knows that something needs attention. It has a list, called the IRQ Polling Table, of every device that might cause an IRQ. For each interrupt, OS-9 runs through the table checking the status register of each device in the table to see if it sent the interrupt. This is classical polling and involves overhead in that it doesn't go directly to the source of the interrupt. The advantage OS-9 has over other operating systems that use polling without interrupts is that OS-9 only needs to poll when something needs attention.

The first entries in the table get slightly faster service than later entries. Devices that need extra fast service can force themselves to the beginning of the table by requesting a high priority when they enter themselves in the table. The F$IRQ service request is used to update the table. Check the example device drivers in the workshop section of this book for samples of the F$IRQ SVC in action.

MASKING INTERRUPTS

Sometimes it is important to execute a block of instructions without interruption. This is almost never a concern for regular programs, but for parts of OS-9 it is a real issue. When a device driver is in the middle of updating a queue, it can't tolerate another version of itself messing with the same queue. If you write a system with two or more processes writing into the same data module, you will need to prevent interruptions during some blocks of code.

The most elegant way to protect a sequence of instructions is with the 6809 instructions that read, modify and write memory all in one instruction. These instructions can't be interrupted even by a separate processor or DMA device. They are inc, dec, com, asl, asr, neg, rol and ror. I usually use the inc/dec pair. They change memory and set the condition code in a convenient way. No item in the list of instructions that read, modify and write as a unit will mask interrupts, but these instructions do modify memory without leaving an opening for an interrupt.

If you need to protect several instructions you can mask interrupts by setting two bits in the condition code register. The sequence is:

```
orcc #IntMask
...
andcc #intMask
```

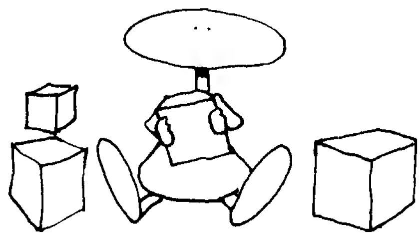
The ellipses represent instructions that can't be interrupted except by a NMI or reset. If you must use this trick, keep the number of protected instructions to a minimum. Don't use any OS-9 service requests; they are likely to unmask interrupts for you. When interrupts are masked, OS-9 can't service interrupts. I/O comes to a halt. In some systems, even the clock falls behind.

SUMMARY

In this chapter you have learned something about what interrupts are and why they are important to OS-9. The underlying mechanism for multitasking came to light, and tricks for protecting "critical sections" of code from interruption were discussed.

CHAPTER 29

modules



In this chapter we go into considerable depth about OS-9 modules.

WHY THERE ARE MODULES

One of the first things OS-9 does while it is booting up is to check through memory for ROM (Read Only Memory), RAM (Random Access Memory) and, in some systems, damaged memory. Undamaged RAM is distinguished by being able to store data. A byte can be saved into RAM and read back intact. To make certain the byte isn't ROM that just happens to contain the test value, OS-9 does this test twice with different values.

ROM always contains the same data regardless of what is written into it, but locations that don't have RAM or ROM can act just the same way. The problem that faced OS-9's designers was how to distinguish ROM from non-memory. Module structure makes useful memory distinct from junk. That is the first use OS-9 makes of modules, but they are also an important part of the overall design of OS-9

HOW MODULES ARE IDENTIFIED

Verifying that a block of memory contains a module has to be done accurately. It would be catastrophic if OS-9 were inclined to find modules where there was only randomness. Since it is frequently employed to identify modules, the process must run fast. The process OS-9 uses is something of a compromise. It

verifies a module in stages. The first two stages will quickly reject almost anything that isn't a module. The last stage is slow, but careful.

The beginning of a module is marked with a special two-byte code, Hexadecimal 87CD. It is possible that this code could occur somewhere other than the start of a module. To make certain that it has identified a module, and to learn more about the module (if that's what it is), OS-9 verifies that the module fits a specified form.

The first nine bytes in a module are called the module header. The header contains the sync bytes ($87CD) and information about the module: its length, name, use, type, attributes, and revision level. After these values comes the second check on the validity of the module, the Header Check byte. The Header Check byte contains the one's complement of the vertical parity of the other eight bytes in the module header.

THE HEADER CHECK

The Header Check is a simple form of CRC (Cyclic Redundancy Check). The vertical parity is taken by *exclusive or'ing* together the first eight bytes. The effect of an *exclusive or* operation on two bytes is to leave 'on' (the value '1') only those bits which are on in only one of the two bytes. For example, if the binary numbers A=%01101010 and B=%10101010 were *exclusive or'd* together, the result would be %11000000. Working from the most significant (leftmost) bit to the least significant bit:

The first bit in A is 0 and the first bit in B is 1.
    Since one of them is 1 the result is 1.
The second bit in A is 1 and the second bit in B is 0.
    Since only A's second bit is 1, the result is 1.
The third bit in A and B are both 1
    so the result is 0.
The fourth bit in A and B are both 0
    so the result is 0.
The last four bits in A and B are identical, %1010 in both,
    so the result is 0 for each bit: %0000.

The 1's complement operation reverses the state of each bit in a number. Each 1 becomes a 0, and each 0 becomes a 1. If a number is *exclusive or'd* (XOR) with its 1's complement, the result is all 1's.

A	**= %11001101**
1's complement of A	**= %00110010**
A XOR Complement of A	**= %11111111**

Because of this trick, it is easy to check the correctness of the Header Check. All the bytes in the header, including the Header Check, are *exclusive or'd* together. If the result isn't %11111111, something is wrong. Actually the easiest way to do this is to take

the 1's complement of the vertical parity (XOR) of all nine bytes in the header. This is the same operation that was used to generate the Header Check, except that we include the Header Check with the rest of the header in the calculation. If the result isn't 0, the check fails.

The type of CRC used to calculate the Header Check is only able to catch one-bit errors in the header. If the Header Check is being verified over random data there is one chance in 256 that it will accept the data as a valid header. If it is run on a damaged header, it will be able to detect some problems, but, if two bytes are damaged, verification of the Header Check can miss the problem. If the third and fourth bytes in the header are %01111000 and %01010011, their *exclusive or* would be %00101011. There are many other pairs of binary numbers that can be *exclusive or'd* with each other to give the same result: %00000000 and %00101011 are a simple example. If only one byte is changed, the Header Check will not verify, but if two numbers are changed, there is a chance the damage will go undetected.

THE MODULE CRC BYTES

It isn't likely that the Header Check and the sync bytes will be correct by chance, but, even if they are, there is one more check which will be made before a block of memory is considered a module. OS-9 keeps a much more sophisticated three-byte CRC check of the entire module. The check is run, from the sync bytes on, for the length given in the module header. The CRC algorithm used will detect any reasonable form of damage, and the chances of it checking out over random data are one in about 16 million. Taken together with the chances of the Header Check verifying on random data, the probability of mistaking junk for a module is only about one in four billion.

There is no need to know the CRC algorithm. It is always best to use the code in OS-9 to generate and check CRCs via the F$CRC and F$VModul system service requests. However, the mark of a good systems programmer is curiosity about just that kind of trivia; so, here are some details about CRC calculation. For those of you who are content to let OS-9 handle this stuff, it is perfectly safe to skip ahead.

HOW THE MODULE CRC WORKS

Cyclic Redundancy Checking (CRC) is an algorithm for error detection in blocks of information. It is more effective for detecting errors than for simple parity checking, but substantially harder to do. In the CRC algorithm, the entire module to be checked is treated as one continuous stream of bits, a large binary number. First, the number is shifted to the left enough to leave space for the CRC code at the low-order end (in the case of OS-9 modules, a three-byte left-shift). The CRC code is the remainder after this number is divided by the "generating polynomial" using mod-2

division. (All operations are in base two, no borrowing or carrying.) The check bits are appended to the end of the module when the module is generated.

When the CRC algorithm is run on a bit stream including the CRC code, the resulting code will be zero. Perhaps an example using standard decimal arithmetic would help (though, in fact, CRC is trickier in decimal).

> If the generating polynomial is the number 13, the CRC code for the number 275,101,712 is 5.
> 275101712 must be shifted left two decimal digits giving 2751017200.
> Dividing by 13 gives 2116167015 remainder 5.
> Subtracting 5 from the original number gives 27510171195.
> Running the CRC algorithm on 27510171195 gives a CRC code of
> 0 because 2721017119500 is perfectly divisible by 13.

You probably noticed that the result of the CRC calculation in the decimal case wasn't the original number followed by the CRC code. When the operation is done in binary mod-2 everything works out smoothly. One important thing to notice about mod-2 arithmetic is that addition and subtraction give the same result. Since there are only the digits 0 and 1, and there is no carry or borrow:

1+1=0 and 1-1=0
1+0=1 and 1-0=1
0+0=0 and 0-0=0

Because of this peculiar behavior, subtraction is a useless operation in proper CRC calculation. Bearing this in mind let's go a little deeper into the math:

Let the module be represented by M
Let the generating polynomial be represented by G
Let the number of bits in the CRC code be k

X = M shifted left k bits

$$\frac{X}{G} = Q + \frac{R}{G}$$

where Q is the quotient from the division and R is the remainder.

$R = X - Q*G$
or, since addition and subtraction are the same,
$R = X + Q*G$

The module with the CRC code attached is
$V = X + R$

Since
X + R = Q*G
V is evenly divisible by G.

The algorithm actually used in OS-9 is slightly different from the standard CRC algorithm. First, since division of large numbers is slow, OS-9 uses a special trick for finding the remainder of mod-2 division that uses mostly shift and *exclusive or* instructions. It also differs from the normal CRC algorithm in that the initial value for the CRC accumulator is $FFFFFF in OS-9 instead of the normal $000000, and the CRC code is complemented before it is used. The result of all the changes is that the CRC code for an intact module including CRC should be $800FE3, the CRC generating polynomial, instead of $000000.

The algorithm used for CRC generation in OS-9 is as follows.

```
    Initialize the CRC accumulator:
        CRC[1] = $FF
        CRC[2] = $FF
        CRC[3] = $FF

For each byte from the beginning to the end of the module
        Let B = the byte
        Let D = B XORed CRC[1]

        /* shift left 8 bits */
        CRC[1] = CRC[2]
        CRC[2] = CRC[3]

        E = D shifted left six (E is a two byte quantity)
        /* add E to CRC using mod-2 addition */
        CRC[2...3] = CRC[2...3] XOR E

        /* Calculate the mod-2 sum of bits 8,7,5, and 1 in D */
        D = (D shifted left 1) XOR D
        D = (D shifted left 2) XOR D
        D = (D shifted left 4) XOR D
        /* If the result of the addition is 1 */
        If the high order bit of D is 1
            CRC[1] = CRC[1] XOR $80
            CRC[3] = CRC[3] XOR $21
```

GETTING AROUND CRC PROTECTION

Every module in memory is validated once before it is placed in the module directory. The validation takes place during bootstrap for ROMed modules, and while a module is being loaded from disk for other modules. It is fortunate that OS-9 doesn't reverify the validity of modules once they are in the module directory because there are many occasions when you will want to modify a module in memory, and generating a new CRC each time a modification is made might be slow work.

Debug may be used to modify a module in memory. It is commonly used to apply patches, and make *ad hoc* modifications to various modules. Changes made by Debug cause the CRC value for a module to change without actually altering the CRC bytes. If the CRC for modified modules were reverified, the changes would cause the module to be rejected for bad CRC.

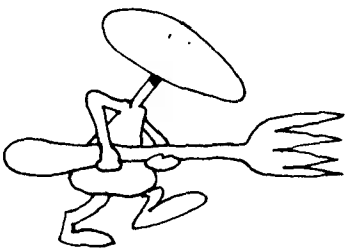
Since a module is safe once it is in the directory, modules can be modified, either because of special circumstances (Debugging) or as a matter of course, as is sometimes done with data modules.

WHAT TYPES OF MODULES ARE THERE?

The high-order nyble (four bits) of the type/language byte in the module header can be used to specify any one of 15 module types. The low-order nyble specifies one of 16 languages. These values are used by the OS-9 kernel and the shell to place the module in the right place and do the right things with it.

Only seven of the available 16 module languages have been defined. Of those seven, only four represent languages that are actually used; and of those four, only two have any real effect on the function of the module. The language specification doesn't necessarily reflect the programming language used to generate the module. It indicates the actual language in the module. If someone ever comes out with a C compiler that generates BASIC09 I-Code, the module will be tagged as I-Code, not C-Code.

The primary use of the language specification is by the shell. When the shell prepares to execute a program, it checks the module's language. If the language is Object code it forks the module, but if it is BASIC09 I-Code, it forks RunB to run the I-Code. This makes it appear that the I-Code file is being directly executed.



The Type specification might be better named Usage. The type nyble tells what the module will be used for. Most non-system modules are programs, but not all. If you write a floating point package as a module that programs can link to when they need floating point services, the floating point module would probably be of the Subroutine type.

Subroutine modules are a feature of OS-9 that hasn't caught on as well as it deserves. The better programming languages permit a programmer to build a program a piece at a time. These pieces are combined by the language or a link editor to form one program which can be loaded and executed. The reasons for modular design have been exhaustively discussed in the Computer Science literature. Let's just say that most people agree that it is a good way to create software. If the pieces (called modules) used to build a program are written generally enough, they can be

collected until programs can be built largely from existing modules. In OS-9, modules need not be passed through a link editor in order to be used. The linkages between the modules can be made as the program is executing by using the OS-9 F$Link and F$Load SVCs.

If a program is written as one large module, changes to that program will require that the entire program be recompiled. If the program is written using modules that are bound to one another by a link editor, then only the modules containing changes will need to be recompiled. The link editor, however, will need to be run again. If the modules are bound while the program is executing, then only the modules that need changes need be recompiled, and no link edit need be done. The old modules must be replaced with the new ones on disk or in memory, and that's that. Sometimes, modules in a running program can be replaced without stopping the program. I don't know of any way to rig a program so that modules actually being executed can be replaced; but with careful (slightly inefficient) programming, a program can be designed so that any module other than the one being executed can be replaced on the fly.

Four special module types are used for OS-9 itself. The OS-9 kernel modules, OS9P1 and OS9P2, together with IOMAN, are System modules. The file managers, Pipeman, SCFMan and RBFMan, are all File Manager Modules. Modules like ACIA, PIA, G68, and CCDisk are device drivers, and device descriptors like TERM and D0 have their own type.

The system module types inform OS-9 that those modules belong in the system area, and give an extra check that a module is the right one.

Two module type/language combinations are special in that they can't be executed. The device descriptors don't contain any executable instructions, just lots of information. There is a provision for non-executable modules other than Device Descriptors. The Data module type can be used for any module that you don't intend to have executed. They are used to store global data and configuration information for user programs. They have even been used as an exotic way of making the I/O address range available to user programs in OS-9 Level Two.

THE MODULE REVISION NUMBER

The byte after the type/language byte in the module header is the attributes/revision byte. The revision number has no influence on what is done with the module once it is loaded — revision two is treated just like revision one. It is only important during the actual loading process. A module already in memory can't be replaced by another module loaded from disk unless the new module has a higher revision number.

Few modules under OS-9 are not reentrant. All the operating system modules are, and all the high level languages generate strictly reentrant code. A reentrant module can be used by several different processes at the same time without any of them knowing that they are sharing the module. The only requirement for this (under OS-9) is that the module must not alter itself or reference any fixed memory locations. Even referencing fixed locations can be done if you are careful.

Reentrant modules are a tool which can save large amounts of memory, but only if they are well used. Some modules, like BASIC09, are frequently used by more than one process at a time. This saves about 24K for each process, after the first, because only one copy of BASIC09 needs to be loaded for all the processes. Most modules aren't as widely used as BASIC09. Most single module programs are too specialized to be of general interest. These programs are probably full of code that would be useful to other programs, but there is no way to share it.

A well-designed modular program is built of a set of modules, each as independent of the others as possible, and each performing one task or a closely related group of tasks. Modules that are part of a well-modularized program are very likely to be useful to other programs. If the modules are reentrant, there will only need to be one copy of a module in memory, regardless of the number of programs using the module.

It would be nice if OS-9 came with a well-developed library of service modules. Perhaps, by the time this book is available, that kind of library will be available. There is room for a module — perhaps several modules — for floating point support, for conversions and print formatting, and for string functions.

A module called GoToXY is often used to interface a program to a terminal. Each person using a program that needs GoToXY must find or write a GoToXY that fits his terminal. If he changes to a new brand of terminal, only the GoToXY module need be changed.

Several types of service modules are already supplied by Microware — the operating system itself. OS-9 is built of modules. The services offered by these modules are accessible through the Supervisor Service requests.

Modules are the way OS-9 stores program code, operating-system static tables, and anything programmers want to stuff into data modules. The module header tells OS-9 enough about a module's contents so that OS-9 is able to treat them as rather-solid objects, distinguished from its fluffy treatment of data areas allocated by processes. You can get the location of any module in memory from OS-9. There is no way to find out from OS-9 where a process's data is located.

OS-9 maintains a list of all the modules in memory, with information about each one, called the Module Directory. In OS-9 Level Two each entry in the Module Directory is eight bytes long. The Module Directory entries each contain a pointer to a DAT image for the home address space of the module, the size of that address space, a pointer to the module header in its home address space, and a link count that records the number of times the module has been linked minus the number of times it has been unlinked. Each of these fields is two bytes long.

The Module Directory entries in OS-9 Level One are only four bytes long. They contain a two-byte pointer to the module, and a one-byte link count. The fourth byte is unused.

The Module Directory entries for OS-9 Level One are shorter and simpler than the entries for Level Two because of Level Two's memory management features, and because Level One is expected to run fewer processes. Each address space in OS-9 Level Two is described by a DAT pointer, which controls the way the DAT (Dynamic Address Translator) will map blocks of memory for that address space. The DAT pointer also controls the number of bytes in the address space. Each module directory entry contains the DAT pointer and length fields to describe the home address space of the module. An address space can contain more than one module; so, the entry also contains the address of the module within the address space. The link count in a Level Two directory entry is twice as long as the link count in a Level One directory entry partly because it is more reasonable for a link count to exceed 256 in Level Two, and partly because the algorithm for finding Module Directory entries requires that their length be a power of two.

The ability to use an address space as home address space for more than one module is an important memory-saving feature of OS-9 Level Two. Since memory must be allocated in blocks that can be mapped with the DAT hardware, OS-9 is usually forced to hand out memory in blocks of two or four kilobytes (depending on the DAT hardware). An address space must contain at least one block of memory. If each module had to have its own address space, each module would consume at least four kilobytes on most systems. Thirty-two small modules would require at least 128K. The DAT pointer is used by OS-9 Level Two to maintain clusters of modules. If you load a group of modules from one file, they will all be in a cluster and share an address space. As long as the link count for any module in the block is not zero, the entire block will remain in memory.

This is a useful feature that must be used with care. If several small modules are collected in a file and loaded together, they can all share an address space, but, of course, there is no way to release part of the address space; so, there is no point in removing

any module in the address space from the Module Directory until the entire address space can be freed. If a process links to any module in the group, the entire bunch comes with it. OS-9 maps the entire address space containing the desired module into the requestor's address space. The best way to cope with both of these difficulties is to moderate the size of clusters of modules. The best size is one DAT block of memory (two or four kilobytes, depending on the DAT).

HOW MODULES ARE GENERATED

The easiest and most common way to generate modules is with a programming language. The BASIC09 pack instruction turns BASIC09 procedures into subroutine modules of BASIC09 I-Code. Languages that compile to native code produce program modules of 6809 object code. Assemblers are also used to produce program modules, but, assemblers being what they are, you can produce any type of module with them.

There are two directives in Microware's standard assembler that are responsible for generating modules. The MOD directive generates the module header, and must come before any constants or operations in the program. The EMOD directive generates the CRC code for a module. It must be the last statement in a module. Other assemblers, such as Microware's RMA (Relocating Macro Assembler), have similar mechanisms for generating module headers and CRC codes.

The MOD statement accepts as arguments the values of the fields in a module header:

1) The first argument is the length of the module. It is possible to figure out this number yourself and insert it here, but it's a lot of work and not necessary. The length needed here is the number of bytes in the entire module, including the header and CRC bytes. The easiest way to find this number is to include an EQU statement right after the EMOD statement, something like this:

```
 EMOD
ModLen equ *
```

The '*' in the EQU statement is replaced with the program address counter for that location in the program. Since the program address counter counts from zero at the beginning of the module header, its value after the EMOD statement is the length of the module.

2) The second argument is the offset to the name of the module. This value can also be calculated by the assembler. The way to do it is to define the name of the program as a constant somewhere in the program, and put the name assigned to that constant in the MOD statement. For example:

MOD ModLen,ProgName...
ProgName fcs /Example/
Edition fcb 1

The assembler will use the address of ProgName in the MOD statement. The program name should be defined using a FCS statement, rather than an FCC, so OS-9 will be able to find the end of the name. The Edition byte following the name isn't necessary, but it's a convention. If you don't put a one-byte field here, indicating the version of the program (or whatever else you like), commands like IDENT that print the edition number will use whatever falls in that spot as the edition. Nothing important relies on this value, but it is easy to include and it makes your programs consistent with most other OS-9 programs.

3) The third argument is the value to use for the type/language byte. There isn't any shortcut for filling this field in. It is best to use the names assigned to types and languages in the OS9Defs file. A module statement like:

MOD ModLen,ProgName,$11,...

will work fine, but it is easier to understand if it's written like:

MOD ModLen,ProgName,PRGRM+OBJCT,...

If you elect to use names for languages and types, be sure to include the OS9Defs file in your program.

4) The fourth argument is the value to use for the attribute/revision byte. The only attributes now supported are Reentrant and write protected — on most systems only Reentrant. The revision is a number that permits you to create a module that will replace a module already in memory. The MOD statement so far looks like:

MOD ModLen,ProgName,PRGRM+OBJCT, REENT+1,...

5) The next two arguments differ according to the type of module being assembled. Some types of module don't use them at all. For program and subroutine modules these arguments are execution offset and permanent storage size. The assembler can calculate both values with a little help. The execution offset is the offset to the first instruction in the program. It can be found by assigning a label to the first instruction in the program and using that label here. The permanent storage size needs a trick similar to the one we used to find the program length. A skeleton program including all the statements and tricks needed to generate a module header and CRC would look something like:

```
 IFP1
 use /D0/DEFS/OS9Defs
 ENDC
 MOD ModLen,ProgName,PRGRM+OBJCT,REENT+1,Start,MemSize
ProgName fcs /Example/
Edition fcb 1
*********
*    Memory and stack space
*
.
.
.
MemSize equ .
Start equ *
    ... The program is here
 EMOD
ModLen equ *
```

MemSize is equated to ' . ', which represents the current value of the program data counter. Since the program data counter, like the program address counter, starts at zero at the beginning of the module, and unlike the program address counter is incremented only for RMB instructions, the

MemSize EQU .

instruction will assign the size of the data memory allocated so far to the symbol MemSize. Another trick I slipped in this example is the way I used the

use /D0/DEFS/OS9Defs

statement. By putting it between IFP1 and ENDC, I prevented the assembler from looking at the OS9Defs file during its second pass. Not only does this speed up the assembly, it also prevents the contents of OS9Defs from being printed out with the program, or even being assigned line numbers.

SVCs THAT DEAL WITH MODULES

The F$LOAD SVC reads modules from a disk file. It needs the pathlist of the file to read from. The documentation may say that it also needs the type/language byte for the module you are looking for, but it is iqnored. It loads all the modules in the specified file into memory and puts them into the module directory, but only the first module in the file has its link count incremented. The F$LOAD service request returns the same values in its registers as a F$LINK on the first module in the file would have; that is, the Module type/language byte of the first module loaded in accumulator-A, the Module attributes/revision byte in B, a pointer just past the path list in X, the address of the module entry point in Y and the address of the module header in U. Note that this service request alters U; in most programs you'll have to push U on the stack before this call and pop it off afterwards.

There are four SVCs that return the address of a module which is already in the module directory.

The most common tool used to locate a module in memory is the F$LINK SVC. It takes, as input, the address of a string containing the name of the module you want to link to and the type/language of the module. The name of the module must start with a '/' or an alphabetic character. It can be terminated with the OS-9 standard ending — high-order bit on in the last byte (use fcs to do this) or with any non-alphanumeric character. If a type/language is specified by loading its value into the A register before calling F$LINK, link will only find a module that matches both the requested name and the requested type/language. If you don't want to specify one of these values, then leave it as zero in the A register. For example, the hex value $10 will match a program module in any language.

The type/language and attribute/revision bytes for a module found by the F$LINK SVC are returned in the A and B registers. The X register is advanced past the module name it pointed to before the SVC. The address of the Module's header in your address space is returned in U, and the entry address in your address space in Y. All that fuss about address spaces is only important if you have Level Two. Under Level One all processes share the same address space.

F$ELINK is a system mode service request that is only available under OS-9 Level Two. It is meant to be used from within OS-9. It needs a pointer to the module directory entry for the module you want to link to. It also needs the attributes/revision value for the module — the System Programmer's Manual says it needs the Module type, but it is incorrect. The attributes/revision byte is used only to determine whether the module is reentrant. The value passed to F$ELINK in register B overrides any value in the module itself.

F$ELINK maps the module into the address space of the process that is currently active according to the D.Proc field in the system direct page. Sometimes this field will point to some process descriptor other than the one you want. This problem can be overcome by temporarily changing the process descriptor pointer in D.Proc. For example, D.Proc can be saved and D.SysPrc used.

F$FMODUL is also a system mode service request for Level Two systems only. It is meant to work with the F$ELINK SVC. F$MODUL searches the module directory for the entry for a given module name and type. The module name is specified by a pointer to the name, and a pointer to the DAT image for the address space the name string is in. Like the F$LINK SVC, it treats zero nybles in the type as wild cards. F$MODUL returns the actual language/type and attribute/revision bytes of the module it finds, a pointer to the Module Directory Entry, and a pointer just past the module name.

F$SLINK is another Level Two system mode service request. It acts like F$LINK, except that it allows a DAT image pointer for the module name string to be specified with the address for the name. It links a module into the current address space, but expects to find the name of the module in another address space. It uses a pointer to the DAT image of the addess space containing the module name string and the offset of the string within that address space to find the module's name string. See the chapter on Level Two Memory Mangagement Internals for details on DAT images.

The F$UNLINK SVC is used to remove a module from an address space, decrement its link count, and, if its link count and the link counts of all other modules in its home address space are zero, release the address space's memory and the Module Directory entries for the modules in the address space. This SVC only requires the address of the module header for the module to be unlinked. Since that address isn't useful for much other than unlinking the module, it is important to remember to save it when the module is linked.

F$UNLINK has the same effect under Level One that it does under Level Two. It reduces the link count of a module, and removes it from RAM if the link count goes to zero.

F$UNLOAD is a user mode service request that is only available under OS-9 Level Two, though I expect it will be added to OS-9 Level One some day. It has the same effect as F$UNLINK, but it uses the module's name and language/type to locate the module, instead of using a pointer to the module's header. F$FMODUL is used to find the Module Directory entry for the module, so the language/type byte can contain zero nybles as wild cards.

F$CRC is used to calculate the CRC for a module, or, for that matter, anything else you want a CRC for. It needs the address of the block of data you'd like a CRC for, the length of the block, and the address of a three-byte variable to put the CRC code in. If it isn't convenient to calculate the CRC on an entire module at once, F$CRC can be used on the module in sections, provided that the sections are in order, starting with the first part of the module. The CRC variable will be used to accumulate the CRC value, so it should be initialized to $FFFFFF before the first call to F$CRC, and then left alone until the entire module has been passed through F$CRC.

If you are using F$CRC to validate a module, accumulate the CRC through the entire module, including its CRC bytes. The accumulator will contain the generating polynomial if the CRC code checks out.

If you want to generate a CRC code, run F$CRC over what you have (that is, everything but the CRC) and complement the generated CRC before storing it at the end of the module. Use the COM 6809 instruction on each byte of the CRC code to complement it.

The service request that verifies a module and places it in the module directory is F$VMODUL. It is a system mode request with significant differences between Level One and Two. If you are using Level One, F$VModul only needs one parameter. It takes the address of the new module in X. If you are using Level Two, you need to give it both the DAT image pointer for the address space containing the new module and the offset of the new module within the memory covered by the DAT image.

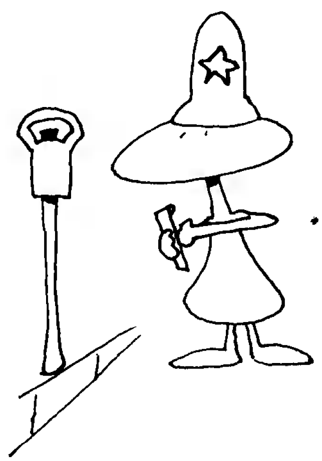
CHAPTER 30

memory management

THE THEORETICAL BASE

In this chapter you will learn about the mechanisms that support OS-9 memory management. The memory management mechanisms used under Level Two represent a substantial extension beyond Level One, so in this chapter we'll concentrate on issues relevant to Level One. The DAT and other issues that only apply to Level Two are mentioned in this chapter, but an in-depth treatment is saved for the next chapter.

Any operating system that permits more than one program to run at a time needs a way of dividing the system's memory between the programs. Both OS-9 levels one and two use rather sophisticated memory management schemes. Instead of jumping right into OS-9 in this chapter, I'll start with some simpler memory management schemes and work up to OS-9's techniques gradually.



Managing a computer's memory is a lot like managing on-street parking. A system's memory, like curb space, must be divided up among users. It is good to use memory (or curb space) as efficiently as possible, but keeping the amount of supervision required to a minimum is also important.

Of course, the simplest way to handle system memory is to ignore the problem. Operating systems that only deal with a single process don't need to concern themselves with memory. There are many examples of operating systems that don't provide any memory management facilities, including PC-DOS, CP/M, and FLEX.

FIXED PARTITION MEMORY

Another simple way to allocate memory is to divide it into several regions called partitions and allocate a partition to each process. The partitions are set up when the computer is started and not changed except by restarting the operating system. Because of the permanent nature of the partitions, this method of memory management is called Fixed Partition memory allocation. This is analogous to the most common way of allocating parking space — marking off parking places with lines on the pavement and placing one car in each place.

It is simplest if each partition is the same size, but each partition can be different. If they are all the same size, a process can either fit in any partition, or none of them. Since each partition is equally useful, the operating system can assign them any way that's simple. Partitions of a variety of sizes allow processes that require lots of memory to be accommodated without having to make all the partitions big enough to hold them. The analogy in the parking world would be to have several different sized spaces. Some suitable for cars, others for trucks.

This enhancement to the partitioned memory system causes lots of trouble. If all the small partitions are full, can a process use one that's far too large for it, or should it wait for a small partition to free up? How many different sizes of partition should be used to get the best possible use of memory? Think of the parking situation again. If all the car slots are full, should a car be permitted to use a truck place? Is it a good idea to have special motorcycle places? How about compact car parking spaces? Remember that small spaces are efficient ways to store small vehicles, but they will be entirely wasted if small vehicles don't need to park.

An operating system that uses partitioned memory is easy to write, but it tends to waste memory. Attempting to fix the problem makes this method complicated without fixing anything.

There are some special cases where partitioned memory is fine. Many operating systems set aside a special area in memory for small utility programs. That is a trivial example of partitioned memory. It is wasted space much of the time, but it isn't a large partition so the waste is minimal. And, it is especially useful for running programs like print spoolers that are meant to be tucked out of the way.

DYNAMIC ALLOCATION

If fixed partitions aren't good enough, memory can be allocated in suitable chunks as it is needed. This is a good idea, but it's not as easy as it might sound. Let's move right to the parking problem. This system is like having a parking attendant who directs vehicles to the right spot without any lines on the pavement. If there are a variety of different sized vehicles, the attendant

should be able to pack them much more efficiently than they could be with pre-marked places.

Things look very good as the first batch of cars and trucks are parked, but after a few leave and others arrive trouble starts to appear. Say the street is filled up end to end, then five small cars leave from five separate locations. Now a small truck arrives. It is small enough to fit in much less than the amount of space just vacated by the five cars, but, since the five empty slots are in five different places, they are useless to the truck, which needs to wait for another truck, or a few cars parked next to each other, to leave. If a truck leaves and a car gets to the place first, it will take up some part of the space. That uses up a space big enough for a truck. Most of the space is still there, but it isn't any good for the next truck to come by.

There are two standard ways of managing memory when it is isn't partitioned in advance. They are called first-fit, and best-fit.

FIRST-FIT ALLOCATION

First-fit allocation is the simplest method. The operating system chooses the first block of memory at least as big as the amount requested. It allocates as much of the block as required leaving the rest as a smaller, unallocated block. Unfortunately, this tends to use up big blocks of memory, leaving lots of little chunks that can only be used to satisfy small requests.

BEST-FIT ALLOCATION

Best-fit requires the operating system to do more work, but it does a better job of keeping large blocks of memory for programs that really need them than first-fit allocation does. In best-fit allocation, the operating system scans through all its available memory looking for the block that fits the request with the least memory left over. This method leaves slivers of unallocated memory around, but it preserves large blocks of memory by refusing to use them as long as any smaller blocks will do the job.

There are other methods. The oddest one I know of is worst-fit allocation. In this method, allocations are always made from the largest block of storage. The reasoning is that the fragment of the large block that is left over after the allocation is made will be larger, and therefore more useful, than it would be if the allocation were made out of a smaller block. My intuition is that this policy amounts to punishing large blocks of free memory. The result probably is a lot of roughly equal-sized blocks of free memory.

Most people use first-fit to manage on-street parking when there is no attendant. Those whose parallel parking skills aren't so good lean toward a worst-fit method — perhaps that's why on-street parking is usually partitioned. If drivers park in the first space they see that is long enough for their car, they are using

first-fit. Those who will go out of their way to find a large space are using a modified worst-fit algorithm.

Best-fit probably wouldn't work without an attendant. It would require each driver to check each empty parking place in town and pick the spot that most precisely fit his car.

OS-9 LEVEL ONE MEMORY MANAGEMENT

To bring reality in for a moment, OS-9 Level One uses first-fit allocation to manage its memory. Microware added a special twist by having module storage and system memory requests start from high memory, and data storage allocated in low memory. You can watch it in action by starting several programs running in background. The easiest program with which to use up space is SLEEP. MFREE reports on where free space is located, and MDIR can list the addresses where modules reside. There is no command in OS-9 Level One that directly reports the data space associated with each process, so this must be inferred from MFREE's output. Try this with a OS-9 Level One system:

> First use the MDIR E command. Look through the first column in its output for the second-lowest module address. On my CoCo the lowest address was B300 for MDIR. The next lowest address was BE00 for CCDISK. As likely as not the addresses will be different on your system.
>
> Next use IDENT to collect information about a few modules. The examples below may not match your results exactly because you may have more recent versions of these modules than I do.

```
OS9:IDENT MDIR -X

HEADER FOR:  MDIR
MODULE SIZE: $01A6   #422
MODULE CRC:  $C459BC (GOOD)
HDR PARITY:  $8F
EXEC. OFF:   $0066    #102
DATA SIZE:   $0127    #295
EDITION:     #03      #3
TY/LA AT/RV: $11 $81
PROG MOD, 6809 OBJ, RE-EN

OS9:IDENT SHELL -X

HEADER FOR:  SHELL
MODULE SIZE: $04FA    #1274
MODULE CRC:  $59ECC8 (GOOD)
HDR PARITY:  $D6
EXEC. OFF:   $003D    #61
DATA SIZE:   $02B5    #693
```

```
EDITION:        $14       #20
TY/LA AT/RV:    $11 $81
PROG MOD, 6809 OBJ, RE-EN
```

```
OS9:IDENT SLEEP -X
```

```
HEADER FOR:     SLEEP
MODULE SIZE:    $004D     #77
MODULE CRC:     $610935 (GOOD)
HDR PARITY:     $65
EXEC. OFF:      $0013     #19
DATA SIZE:      $0200     #512
EDITION         $01       #1
TY/LA AT/RV:    $11 $81
PROG MOD, 6809 OBJ, RE-EN
```

```
OS9:IDENT MFREE -X
```

```
HEADER FOR:     MFREE
MODULE SIZE:    $176      #374
MODULE CRC:     $65997C (GOOD)
HDR PARITY:     $5F
EXEC. OFF:      $006A     #106
DATA SIZE:      $021F     #543
EDITION:        $05       #5
TY/LA AT/RV:    $11 $81
PROG MOD, 6809 OBJ, RE-EN
```

We're going to be using MFREE, MDIR and SLEEP a good deal; so, to save time and prevent some confusion, load them into memory:

```
OS9:load mfree
OS9:load sleep
OS9:load mdir
```

Now, have a look at the system's current memory distribution.

```
OS9:MFREE

ADDRESS   PAGES
-------   -----
B00-AEFF  164
B400-B5FF   2

TOTAL PAGES FREE = 166
GRAPHICS MEMORY NOT ALLOCATED
```

MFREE is displaying the free memory in a system running the sysgo, shell and mfree. The OS-9 System Programmer's Manual shows that OS-9 uses memory from $0000 to $03FF. Sysgo uses one page from 400 to 4FF.

The shell needs 694 bytes for data, which rounds up to three pages (from \$0500 to \$07FF). MFree needs 543 bytes for data, which also rounds up to three pages (from \$0800 to \$0AFF). From 0B00 on up to AEFF is one free block of memory. It appears that LOAD used a page of system memory (allocated from the top of available memory) in addition to the page needed for the LOAD module. It grabbed the memory from B400 to B5FF while I was loading modules into memory, and returned it in time for it to show up as free memory for MFREE. If that memory had been available for module storage when the modules were being loaded, they would have used those pages and wouldn't be free now.

The SLEEP module is already in memory, so each invocation of it will only allocate data memory starting in low memory. To start, try:

```
OS9:SLEEP 1500 #20&
OS9:MFREE

ADDRESS   PAGES
-------   -----
1F00-AEFF  144
B400-B4FF    1

TOTAL PAGES FREE = 145
GRAPHICS MEMORY NOT ALLOCATED
```

First, notice that something allocated another page of system memory from B500 to B5FF. Perhaps it was needed to store the overflow of various descriptors, since I was running a total of four processes (SYSGO, SHELL, SLEEP, and MFREE). There are also another 20 pages allocated from 0B00 to 1EFF — the pages I request in the

```
SLEEP 1500 #20
```

command. If you wait a while for SLEEP to end, you will notice that the memory from 0B00 to 1EFF is returned, but the page at B500 remains allocated. The operating system tables are designed to expand when necessary, and haven't been able to shrink back down yet.

Now, let's confuse issues by starting a bunch of SLEEP's for different lengths of time and with different memory requirements. The best way to do it is to build a file with the list of commands in it and then execute it.

```
OS9:BUILD TMP
? SLEEP 1000 #50&
? SLEEP 10000 #2&
? SLEEP 1000 #40&
```

```
? SLEEP 10000 #2&
? <ESC>

OS9:TMP
&004
&005
&006
&007
```

The memory allocation now is:

```
0000-03FF   System
400-04FF    SysGo
0500-07FF   Shell
0800-0AFF   Free memory where shell(2) was
0B00-3CFF   Sleep
3D00-3EFF   Sleep
3F00-66FF   Sleep
6700-68FF   Sleep
6900-AEFF   Free memory
```

```
OS9:MFREE

ADDRESS    PAGES
---------  -----
6900-AEFF    70

TOTAL PAGES FREE = 70
GRAPHICS MEMORY NOT ALLOCATED

<wait a while>

OS9:MFREE

ADDRESS    PAGES
---------  -----
0B00-3CFF    50
3F00-66FF    40
6900-AEFF    70

TOTAL PAGES FREE = 160
GRAPHICS MEMORY NOT ALLOCATED
```

The actual memory allocation is now:

```
0000-03FF   System
0400-04FF   SysGo
0500-07FF   Shell
0800-0AFF   MFree
0B00-3CFF   Free memory
3D00-3EFF   Sleep
3F00-66FF   Free memory
6700-68FF   Sleep
6900-AEFF   Free memory
```

It might seem strange that there are 50 pages of free memory starting at 0B00, even with MFree running. Remember that running a shell file requires a special invocation of the shell — called shell(2) in the chart of memory allocation — which needs three pages of data space. MFree fits into the space that shell was using.

There are 163 pages free, but if you try to run a program that needs more than 70 pages, the program will not be able to find sufficient memory. Try it.

```
OS9:SLEEP #75
ERROR #207
```

Error number 207 is the "insufficient contiguous memory" error. It is telling us that OS-9 can't find enough memory in one block to satisfy our request.

To verify that OS-9 is using first-fit, try

```
OS9:SLEEP 600 #40&
```

This could fit in any of the three blocks of free memory. If OS-9 is using best-fit, it will land in the 40 page slot in the middle. If it is using first-fit, it will land in the 50 page slot or the 70 page slot, depending on the direction from which it is searching. Using a worst-fit algorithm, OS-9 would pick the 70 page slot. Another MFREE command

```
OS9:MFREE

ADDRESS  PAGES
-------  -----
3300-3CFF  10
3F00-66FF  40
6900-AEFF  70

TOTAL PAGES FREE = 120
GRAPHICS MEMORY NOT ALLOCATED
```

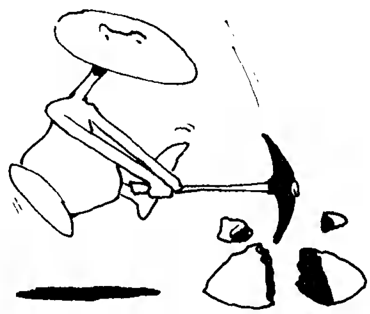
tells us that OS-9 used the memory from 0800 to 30FF for the 40 pages SLEEP needed (the memory from 3100 to 32FF is used by MFREE). This proves that OS-9 uses first-fit allocation.

FRAGMENTATION

Every method of allocating system memory on the fly can be pushed into a situation where there is plenty of memory available, but the memory is divided into so many small blocks that it is useless. If it were only possible to shift the allocated blocks of memory around until all the free space between them was squeezed out to one end, fragmented memory could be made useful again. The process of rearranging memory to make large

blocks of free space is called "garbage collection." BASIC and LISP usually have built-in garbage collection — they pause from time to time and organize their storage. OS-9 doesn't do garbage collection. If you usually run just one or two processes at a time, you probably won't have any trouble with memory fragmentation.

Memory fragmentation is caused by dynamic memory demands. If OS-9 only has to deal with one or two blocks of memory at a time, it can keep things in good order. But if you allocate and release memory frequently, there is real potential for trouble.



One cause of fragmentation is hard to control. The first time a device is opened, OS-9 will allocate static memory for it out of system memory (high memory). If high memory is crowded when you open the device, the static storage could be located in an inconvenient spot. Killing processes and unlinking modules won't make the static storage go away. The best way to avoid the problem is to open your devices early, even if you don't plan to use them until later.

In the example we just went through, the fragmentation took place in data storage, but module storage can have the same trouble. If you use modules heavily — loading them when you need them and unlinking them when you don't — you can fragment memory.

Starting lots of processes (as we did with SLEEP) can fragment memory. Each process allocates some data space. If all processes retain their memory for about the same length of time, any fragmentation tends to heal eventually; at least, it seems under control. The worst situation is caused by a long-running process that has a chunk of memory right in the middle.

There is only one way to de-fragment memory: kill some processes so they release their memory. Then restart them. The restarted processes will get memory at the far ends of memory and leave the prime locations in the middle free. In the example we ran through with SLEEP, the fragmentation situation would have been much improved if we had KILLed the second two-page sleep (process seven). If we had KILLed both of the sleeps with two pages of memory, free memory would have been contiguous.

DYNAMIC ADDRESS TRANSLATION

Memory fragmentation can be a serious problem under OS-9 Level One. Level Two of OS-9 is meant to handle several users and many processes. It also can use far more than the 64 kilobytes of memory that OS-9 Level One can manage. To make more than 64 kilobytes accessible to a 6809 microprocessor, and to handle memory fragmentation, computers capable of running Level Two need Dynamic Address Translation hardware.

Take an imaginary computer with 1024 kilobytes (One Meg-

abyte) of memory and a typical DAT (Dynamic Address Translator). Without the DAT, the 6809 could access the bottom 64K of memory. With the DAT, the 6809 can access any 64K, selected 4K at a time, out of the 1024K and arrange it in any order. If there are several processes running, each can have access to a different part of memory.

Most 6809 programs don't need the entire 64K "address space" of the 6809. The DAT can be set up to give a program as little as 4K of memory. This cuts down on wasted memory.

A 6809 uses 16-bit numbers as addresses. The low-order 12 bits give the address within a 4K block. The high-order four bits select the 4K block. The low-order bits are used directly to address memory, but the top four bits are sent through the DAT first. The four-bit number is used by the DAT to index into a table of eight-bit numbers (called the translation table). The eight-bit number found in the table comes out of the DAT and is used as the top eight bits of a 20-bit address. The processor generates 16-bit addresses. The DAT takes the 16-bit address and generates a 20-bit address. The 20-bit address is used to access memory, giving the CPU/DAT combination the ability to get at 1M (Megabyte) of memory.

A 64K "address space" is made up of sixteen 4K blocks. A minimal DAT only needs to keep a table of 16 bytes of information.

For the simplest systems, all that is required for a DAT is enough very fast memory to store those 16 bytes. The high-order four bits of the address coming out of the 6809 are used as an address in this tiny memory. The number read from the memory that is serving as a DAT is used to extend the low-order 12 bits of the address into a 20-bit address.

The problem with a DAT this simple is that every time you want to change the set of 4K blocks accessible to the processor you have to write new values into the translation table. In a system running several processes, the translation table has to be changed 100 or more times per second. Finding the right "map" to load into the DAT, and loading it 100 times per second, can slow a computer down noticeably. The better DATs keep 16 translation tables. They have a task-select register that is used to choose the right table. As long as no more than 16 address spaces are active, all OS-9 needs to do to switch translation tables is to write the correct number in the task-select register. If there are more than 16 address spaces active, OS-9 has to remember which translation tables are loaded in the DAT, and replace one of them when an address space that isn't represented in the DAT needs to be accessed by the processor.

It is not a good idea to fool with the DAT directly if you're using OS-9. If you change a DAT register without going through the proper procedures, OS-9 won't know what you did. The effect can

be chaos. It is, however, good to know what the DAT is doing and how to take advantage of it.

The DAT maps blocks of memory into the appropriate address spaces. Normally, the blocks mapped into each address space are "disjoint"; that is, each address space has its own memory with no overlaps. This protects processes from interference. No process can read or write memory belonging to another process. If it is important for a process to have access to blocks of memory mapped into another process's address space, this protection can be frustrating. Although the DAT is usually used to protect processes from each other, it can map a block of memory into more than one address space. The block of memory can even appear at a different address in each address space. A block of memory can also be made to show up at several locations in the same address space, or even moved, rather laboriously, from place to place in an address space. All these things can be done with OS-9 service requests. Most of these service requests are system-mode requests, but, for the determined user, the services are there.

VIRTUAL MEMORY

As I write this, there is no virtual-memory version of OS-9. The OS-9, 68K Level Three system that Microware has been talking about will implement virtual memory, so it falls just barely within the scope of this book.

Virtual memory is the next step beyond a DAT. It is needed when programs require more memory than is available on the computer. Virtual memory makes the machine appear to a program to have more memory than it does. The trick is done by keeping some of the blocks (or pages) of memory that appear to be addressable as main memory on secondary storage, usually disk. To oversimplify, the DAT contains the address of a page in main storage, if it is there, or it calls attention to the fact that it has been saved on disk, if that is the case. When the processor tries to access a page that isn't in memory it gets an error called a page fault. The operating system then finds the necessary page on disk, loads it into memory, and allows the process that required it to continue.

LEVEL ONE SYSTEM SERVICE REQUESTS

In user mode there is only one system service request that directly effects memory management: F$Mem. This request is used to set the memory allocation of a process. When F$Mem is called the D register must contain the number of bytes you would like the process to have. If the amount of memory requested is less than the current allocation, OS-9 will decrease the allocation to the amount requested. If additional memory is required and the memory is available, OS-9 will increase the process's allocation. In both cases, the request will return the amount of memory allocated to the process in D, and the upper boundary of the region in

Y. The actual size is useful because OS-9 always allocates memory in pages. Your request will be rounded up to the nearest page. If the amount of memory requested is zero, OS-9 doesn't change the memory allocation; it only returns the size and high bound of the process's memory.

The F$Mem request has several possibilities for error. First, the memory may not be available. Under OS-9 Level One, memory for expansion must be available right above the current allocation. If some other process has that memory, there is no way to get more memory until the other process terminates. Even decreasing the memory allocation can cause trouble. If you try to release memory including the page containing the top of the stack (where the stack pointer "S" is pointing), OS-9 will accuse you of a suicide attempt and return an error.

There are four system-mode system calls that allocate, or free, memory. The OS-9 kernel usually allocates memory using the F$SrqMem system call. This call allocates memory starting in high memory. Unlike the F$Mem call F$SrqMem only allocates new memory. The parameter passed to it in the D register is the amount of new memory required. The only value returned is a pointer to the beginning of the new allocation. The caller is responsible for remembering that memory is allocated in pages. That F$SrqMem makes no attempt to allocate memory that is contiguous with other memory allocated to this process is another important difference between the F$SrqMem and F$Mem.

Memory allocated with F$SrqMem must be returned with F$SRtMem. This call takes the starting point and size of the memory to be returned and returns that memory to the pool of free memory.

Many of OS-9's memory requirements are for blocks of memory smaller than a page. The F$All64 and F$Ret64 system calls manage dynamic tables of 64-byte blocks of memory. The parameter for F$All64 is the base address for the table. If this is the first call for a table, the base address won't be known, so zero can be used. The call returns the block number of the new 64-byte block, the address of the base of the table and the address of the new block. A block can be deallocated with the F$Ret64 call, which takes the base address of the table and the block number as parameters. Memory for the blocks managed by F$All64 and F$Ret64 calls is allocated by F$All64, using the F$SrqMem call. It comes from high memory.

The set of service requests including F$A1164/F$Ret64 is completed by the F$Find64 request. It doesn't affect the amount of memory allocated; it locates a block allocated by F$A1164 based on its block number. This combination of three service requests is enough to completely hide the operation of the 64-byte block-management algorithm from a programmer.

For the curious, this is how it works. The base block for the table contains a list of one-byte pointers. If a pointer is zero, that indicates that no page is allocated for that pointer, yet. If it is non-zero, the value is part of the address of a 256-byte page. The address of each page is a multiple of 256, so the low-order byte of each page address is zero. Only the high-order byte of a page's address needs to be stored.

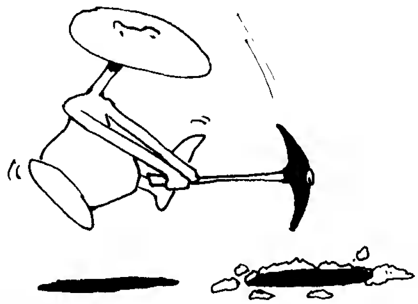
The algorithm for finding a block is a little wasteful of space, but fast. The 64-byte base of the table contains up to 64 one-byte pointers, each pointing to a page containing up to four blocks. A block is located by its block number. The high order six bytes of its number are used to select a pointer from the base table. If that pointer is non-zero, it is used to generate a pointer to the page containing the target block. The low-order two bytes of the block number, multiplied by 64, give the offset in the page of the target block.

This is fast because a block can be found without any searching, but potentially wasteful of space if blocks are allocated and freed in a pathological way. By allocating many blocks, then freeing all but numbers 5, 9, 13, and so forth, the memory allocated for the blocks can be left only one-fourth full. This isn't likely to happen in ordinary usage, and the speed F$Find64 gains from this choice of algorithm is worth the possible wasted space.

A warning: Don't mess with the page table or the first byte of a block!

CHAPTER 31

memory management — level two



In this chapter we will dig right down to the lowest levels of OS-9 memory management. The details of Level Two privileged service requests are pretty tough going. Don't bother to slog through them unless you need the information or can't stand the suspense.

The most important difference between OS-9 Level One and OS-9 Level Two is the way Level Two manages memory. Level Two uses Dynamic Address Translation (DAT) hardware. Dynamic address translation gives a system a way to use lots of memory, even if its processor can only address 64K. It permits each process to run in its own memory (address space) isolated from all other processes. It would seem that, with extra memory and dynamic address translation to handle, OS-9 Level Two's memory management would be much more complicated than Level One's. In a way, it is; but from the average user's standpoint, Level Two is much simpler than Level One.

The only exposure most users get to memory management is through memory requests by the shell:

OS9:asm test #100 * allocate 100 pages of data storage

and application program memory reconfiguration, e.g., the BASIC09

statement. In both cases (and to the underlying programs) the commands look the same under Level One and Level Two. From this point of view, the only ways Level One and Level Two differ are in errors that can be returned from a memory allocation request, and in the amount of memory available to each process.

After operating system memory requirements have been satisfied, there are about 44 kilobytes of memory left for user programs under Level One. If you want to run more than one process, they must divide the memory between them. The OS-9 Level Two operating system draws most of its memory requirements from a special system address space. Depending on a system's DAT hardware, each process can get from 60K to 64K of memory. Most Level Two systems have at least 128K of memory; 256K or more is typical. It isn't unusual to have 16 or more processes running at the same time, making the amount of money you can spend on memory the real constraint on memory-per-process.

ERROR MESSAGES

Under Level One there are two reasons for a failure of a memory request: either you asked for more memory than was available, or memory fragmentation didn't leave a large enough block to satisfy your request. Under Level Two there are also two reasons for a memory allocation failure: either you asked for more memory than was available, or your process has asked for more memory than it could address (more than 64K). Of these, the hardest problem to understand and correct is found only under Level One — fragmentation.

MEMORY MANAGEMENT SYSTEM SERVICE REQUESTS

USER MODE REQUESTS

The F$Mem service request is, by a large margin, the most-used memory management service request for application programs. It works identically under Level One and Level Two.

The F$Link service request isn't a memory management service in the same sense as F$Mem, but under Level Two it has an important effect on memory management. Under Level One, linking to a module doesn't effect memory allocation in any way; mostly, it returns a pointer to the module header and a few values taken from the module header. Under Level Two, linking to a module appears to work just the same way; the request returns the address of the module header and some information from it. A lot is hidden behind this. A reentrant module can be shared by several address spaces. This means that a module can't simply be copied into the address space of each process that wants to use it. In order to conserve memory (and maintain compatibility with Level One),

OS-9 uses the DAT to map the memory containing a module into the address space of each process that links to it.

The F$Load service request loads a module (or group of modules) into the address space of the caller, and also into a special "secret" address space just for that group of modules. Each time a F$Link request for a reentrant module in that group is done, the memory for the entire group is mapped into the requestor's address space. This trick for mapping blocks of memory into several different address spaces is an especially useful one. In the Workshop section (the Daemon program) we illustrate a technique for using shared modules as a path for interprocess communication.

There are four user-mode service requests that are used to cross the boundaries between address spaces. These commands are used by OS-9 commands to display the state of the operating system. One of them is a general-purpose tool for reading any memory in the system.

The MFREE command uses F$GBlkMp to gain access to the memory block map. This control block, like all other OS-9 control blocks, is kept in the system address space where it is normally inaccessible from any user program. It needs a 1024-byte buffer —enough for two megabytes of 256-byte pages. The actual amount used for the map is returned in the Y register. This request also returns the size of a memory block, which can vary from system to system depending on the type of DAT being used. In addition to being the approved way to read the memory block map, this is the only "front door" way of getting the memory block size. F$GBlkMp should be chosen over other ways of getting at the memory map because it is a "front door" access.

The MDIR command uses F$GModDr to get a copy of the module directory. F$GModDr works very much like F$GBlkMp in that it copies a system control block into a user address space, but it returns a pointer to the end of the directory instead of a length (as returned by F$GBlkMp). This service request also returns a bit of arcane knowledge: the address of the module directory in the system address space.

The manual says that the F$GModDr service request doesn't return anything in a register, but takes some strange arguments. The program GetMDir gives an example of the use of F$GModDr. It takes the address of your buffer as an argument, and returns the end of the directory in your buffer and the address of the module directory in the system address space. GetMDir copies the module directory to standard output. The best way to use the program is:

OS9:GetMDir ! dump

Dump will put the output of GetMDir in dump format. This will

let you see what's in there and protect you from strangeness on your screen.

```
                                ttl     Get Module Directory
                                nam     GetMDir
                                IFP1
                                ENDC
  0011                  Type    set     Prgrm+Objct
  0081                  Revs    set     ReEnt+1
  0000 87CD0034                 MOD     GetMEnd,Name,Type,Revs,Entry,MemSize
  000D 4765744D         Name    fcs     /GetMDir/
  0014 01                       fcb     1
       *************
       *  Local Storage
       *
D 0000                  Buffer  rmb     2048
D 0800                  Stack   rmb     200
D 08C8                  MemSize equ     .
  0015                  Entry
  0015 30C4                     leax    Buffer,U
  0017 3440                     pshs    U
  0019 103F1A                   OS9     F$GModDr     Grab Module Directory
  001C 3540                     puls    U
  001E 250E                     bcs     Error
  0020 1F20                     tfr     Y,D          end of MDir in buffer
  0022 3410                     pshs    X            address of start
  0024 A3E1                     subd    ,S++         calc length
  0026 1F02                     tfr     D,Y          length to Y
  0028 8601                     lda     #1           std out
  002A 103F8A                   OS9     I$Write
  002D 5F                       clrb                 clear carry
  002E                  Error
  002E 103F06                   OS9     F$Exit
  0031 A75C6E                   EMOD
  0034                  GetMEnd equ     *
```

F$GPrDsc copies the contents of a designated process descriptor from the system address space to a buffer in a user address space. PROCS uses it to get information about processes. It would also be a way for a process to discover its priority. This service request doesn't return any values in registers; it just places a copy of the requested process descriptor in the buffer.

If these "front door" methods of reading system memory aren't enough to meet your needs, OS-9 provides a generalized tool. F$CpyMem has access to any non-protected byte in the system. Like the other user-mode cross-address space requests, F$CpyMem can't change anything in another address space, but it can look. This service request needs a lot of information. The number of bytes you want to copy and the address of the buffer you want the data placed in are pretty straightforward. The "DAT image pointer" and "offset in block to begin copy" require a little more thought.

The offset is an offset in the address space defined by the DAT image. Since this may not correspond with a real address space, the offsets may need to be adjusted from true addresses. If the DAT image you are using is an exact copy of the DAT image of a real address space, then the offset is the address within that address space. If the contents of the DAT image you use differs from the DAT image of the address space you want to raid, you may need to make some adjustments to addresses.

You'll see the phrase "block offset" often in the OS-9 System Programmer's Manual. This phrase reflects the fact that you'll seldom bother to create more than one block of DAT image. It isn't necessary to define blocks in the DAT image that you don't want to access. There aren't many reasons to want more than a block of data from another address space. Since DAT images for these calls usually contain only one block, the offset within the block is equivalent to an address in the address space defined by the DAT image.

Each block in a DAT image is specified by a two-byte number in the DAT image. If, for example, you want to see the contents of the system direct page (which is located in the first 256 bytes of real memory), you could construct a two-byte DAT image containing zeros, and use an offset of zero. If you wanted a copy of the lowest and highest memory in the system (allowing for a one-megabyte system with 4K blocks), you could use the following assembler statement to allocate a DAT image:

```
DATImage fcb 0,0,0,$FF
```

By using an offset of zero and a length-to-copy of 8K, you could get the data copied into your buffer.

After the four meaningful bytes in the two-block DAT image above, the system will find 28 bytes (60 bytes on systems that use 64 bytes for a DAT image) that just happened to be there. That's fine. There'll be lots of bytes in the DAT image that you didn't want, but they'll be out beyond the area you are going to copy from, so no harm is done.

GetSysMem is a simple program that uses the F$CpyMem service request to get a copy of the first 4096 bytes in the system address space. If you have a system that doesn't use 4096 as its DAT block size, you'll have to lengthen the DAT image or decrease the length of the copy in GetSysMem. The output of the command makes interesting studying, but it's hard to understand as it flies by. Hardcopy output is crucial. The command line is:

```
OS9: getsysmem ! dump >/p
```

```
                ttl   Get low system memory
                nam   GetSysMem
                IFP1
```

```
                                 use    /d0/defs/os9defs
                                 ENDC
      0011             Type      set    Prgrm+Objct
      0081             Revs      set    ReEnt+1
      0000 87CD0039              MOD    GetMEnd,Name,Type,Revs,Entry,MemSize
      000D 47657453    Name      fcs    /GetSysMem/
      0016 01                    fcb    1              edition
      0017 0000        DAT       fcb 0,0
           ************
           *  Local Storage
           *
      1000             BufferS   set    4096           Buffer size
    D 0000             Buffer    rmb    BufferS
    D 1000             Stack     rmb    200
    D 10C8             MemSize   equ    .
      0019             Entry
      0019 308DFFFA              leax   DAT,PCR
      001D 1F10                  tfr    X,D
      001F 108E1000              ldy    #BufferS
      0023 8E0000                ldx    #0
           *  U already points to the start of local memory (the buffer)
      0026 103F1B                OS9    F$CpyMem
      0029 2508                  bcs    Error
      002B 8601                  lda    #1             std out
      002D 30C4                  leax   Buffer,U       point X at buffer
           *  Y already = BufferS
      002F 103F8A                OS9    I$Write
      0032 5F                    clrb
      0033             Error
      0033 103F06                OS9    F$Exit
      0036 980C5A                EMOD
      0039             GetMEnd   equ    *
```

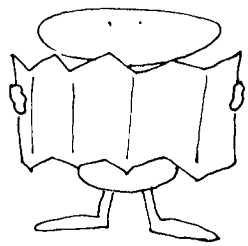
On some systems the F$MapBlk service request is a user-mode request. It is more powerful than the F$CpyMem request because it actually maps blocks of memory into the caller's address space. Since the blocks are made addressable, data in them can be changed. The F$CpyMem request protects data outside a process's address space from damage. The F$MapBlk request leaves all memory in the system open to damage.

F$MapBlk takes two arguments: the number of the first block you want, and how many blocks you want. Some blocks, like block zero (for most system static storage) or the block containing memory mapped I/O, are easy to to find. Outside these special blocks it's a nice piece of work just finding out what block number you want. The trick I like best gets at the block number through some process's process descriptor.

Most of the time, the block of memory you want will be in some process's address space. If you want access to some other process's memory and you know its process id, you can find the block number by requesting a copy of that process's process descriptor (with F$GPrDsc) and picking the block number out of the process's DAT image in the process descriptor.

CHAPTER 32

level two memory management internals



All versions of OS-9 use a memory block map as a crucial part of their memory management scheme. Pointers to the beginning and end of the memory block map are found in the system direct page. The map contains a byte corresponding to each memory block that the system could contain. At least three bits in each byte have significance. The block can be "not RAM," "RAM in use," or it can contain a module. It is impossible for a block of memory to be "not RAM" and "RAM in use," but any other combination is possible. When a memory request is made, the memory block map is searched for memory to satisfy it. The memory block map also protects memory that contains a module from being freed while the module is still in the module directory.

Dynamic Address Translation (DAT) hardware typically requires two types of information: Task numbers, and associated DAT images. OS-9 Level Two's memory management rotates around those values, particularly the DAT image. Each address space requires a DAT image. An address space also needs a task number before it can be accessed, but a task number need not be assigned until it will be used.

Each process's Task number is stored in the P$Task field in the process's Process Descriptor. The Process Descriptor also contains the entire 64-byte DAT image for the process.

MODULE MEMORY

Modules are stored by slight-of-hand. They are mapped into the address space of each process that uses them, but they are maintained in a special address space that only has a DAT image — no task number. A task number isn't needed because the special address space is never accessed. It is strictly a way to keep the modules around when they aren't in any process's address space. Each module directory entry contains a DAT image pointer and the length of the memory block (address space) containing the module. If several modules were loaded together, all the modules will share a DAT image and have the same memory block size. This is reasonable in that they all reside in the same address space.

When a process terminates, it normally releases all the memory that was allocated to it. The memory allocated to modules is released from the address space, but it is marked in the system memory map as being occupied by a module. Memory occupied by a module can only be freed by unlinking all the modules in it until their link counts are zero.

DAT images and module directory entries share an area in the system address space. The module directory entries work their way up from the low end of the block. The DAT images run from the top down. These are special control blocks in that the module directory entries aren't pointed to by anything except temporary pointers within OS-9, and the only lasting pointers to the module DAT images are in the module directory entries. Garbage collection is done when the block of memory used to store these two data structures becomes too fragmented. The garbage collection algorithm compresses all the module directory entries at the bottom of the area and the DAT images at the top. The DAT pointers in the module directory entries are adjusted to point to the new locations of the DAT images as they are moved about.

SYSTEM-MODE MEMORY MANAGEMENT SERVICE REQUESTS

The Level One system-mode memory-management service requests are duplicated in Level Two. The 64-byte memory block management requests are included in that list. They work just the same under both versions of OS-9. The system memory request and return calls also behave like the Level One equivalents.

Things get interesting when we look at the large number of memory management system service requests added to Level Two. The requests can be divided into five groups of requests and two that don't fit into any classification.

The OS-9 System Programmer's Manual refers to a "DAT image offset." This term is a little confusing. The DAT image offset isn't literally the offset within a DAT image. Since block numbers in a DAT image take two bytes each, blocks in an address space would be numbered 0,2,4,... if offsets in the DAT image were used. The offset referred to in this service request is the position of the block in the address space. I'll try to call this value a relative block

number whenever I can. The "block offset" is an offset from the beginning of the block specified in "DAT image offset."

CROSS MEMORY SERVICES

This group of service requests provides ways of getting information from any address in the system. This process is inherently slow; time gets spent fussing with the DAT in the process of transferring each byte. A system with memory-to-memory DMA hardware can move data between address spaces quickly, but that is cheating. Memory-to-memory DMA includes special hardware that can reach any address in the system without using the DAT.

The F$LDABX/F$STABX pair works like the 6809 instructions LDA and STA. Specifically they mean:

F$LDABX
Load A with the byte at offset ,X in the address space of process B.

F$STABX
Store A into the byte at offset ,X in the address space of process B.

They can be used from the system address space to get important bytes (say a buffer of data to output). The program fragment

```
ldb Task1
ldx X
OS9 F$LDABX
ldb Task2
OS9 F$STABX
```

would get a byte from address X in the address space of Task1 and put it at the same address in the address space of Task2.

The F$LDAXY command is for use when you aren't concerned with tasks. It doesn't require you to know which task owns the block you want to access. It takes an offset and a pointer to a one-block DAT image. The one-block DAT image is just the two-byte block number of the block you want data from. The offset must, of course, be to an address within the block specified in the DAT image.

F$LDDXY is a more complicated version of F$LDAXY. It can deal with larger offsets formed by adding D to X:

```
leax D,X
```

This address can be greater than the size of a DAT block. F$LDDXY can handle DAT images that define more than one block, but be sure to give it a large enough DAT image to include the offset given by the sum of D and X.

The F$LDABX and F$LDAXY families use different tricks. F$LDABX uses the task-switching feature of a DAT. It sets the DAT to the task in B, loads the byte you want and switches the DAT back to the system task. This has to be done from the top 256 bytes of memory, which are mapped into every address space; otherwise, when the task was switched, the code doing the F$LDABX would be mapped out of the address space. F$LDAXY works by altering the map in the DAT without changing the task number. The block you want is mapped into the first slot in the system address space, the byte you want is recovered and the original block is mapped back into the first position. This trick relies on the fact that OS-9 always uses block zero in the first slot in the system address space. Since only one block in the DAT is changed, and that block doesn't contain the code or any of the data being used, the code for F$LDAXY does not have to be located in the top 256 bytes of each address space.

The final cross-memory service request is F$Move. This service request requires that every available register be loaded with information. It needs the source and destination task numbers, addresses in each address space to start the move, and the number of bytes to move. It moves data from one address space to another. This could be done with repeated F$LDABX/F$STABX requests, but, by staying in high memory and moving bytes without the overhead of lots of service requests, the F$Move request moves blocks of data as quickly as possible. If a system has memory-to-memory DMA hardware, the F$Move request is particularly efficient.

DAT IMAGE CONTROL

Three service requests are dedicated to managing DAT images. The F$AllImg request is used to increase the number of blocks of memory in a DAT image. A DAT image isn't normally full of live memory. Unless a full 64K of memory is allocated to the address space controlled by a DAT image, several blocks in it will be marked as "Free memory." Free memory is indicated by a special block number that is used for DAT blocks that aren't in use. F$AllImg is given a pointer to a process descriptor. It modifies the DAT image in that process descriptor such that the image has the requested number of blocks of memory starting at the specified relative block.

Let me take the F$AllImg request a little more slowly. F$AllImg is given a pointer to a process descriptor. It is also given a starting relative block number (A) and a number of blocks (B). OS-9 runs through the DAT image starting at the specified block (A), making certain that the next (B) blocks are allocated. If any of the blocks are "Free memory," F$AllImg gets an unused block from the system memory map to fill that block. If all the DAT image blocks that F$AllImg scans through are allocated, it doesn't do anything.

In a system with 4K DAT blocks, the program fragment

```
ldx D.Proc
lda #8
ldb #2
OS9 F$AllImg
```

would make certain that memory from $8000 to $A000 was allocated in the address space of the process who's process descriptor was pointed to by D.Proc.

F$DelImg releases blocks of memory from a process's DAT image. It takes the same arguments as F$AllImg, but has the opposite effect. It runs through the specified blocks in the process's DAT image and returns them to the system's free pool.

F$SetImg copies a block of memory into a process's DAT image. You give it a pointer to the process descriptor, the starting relative block number in that DAT image where copying begins, and the number of blocks to copy. The purpose of this request is to merge two DAT images. If you want to map two blocks from one address space, A, into another, B, you must find two contiguous free blocks in address space, B, and use F$SetImg to copy the selected two blocks worth of DAT image from A's DAT image to the free space in B's DAT image.

All three DAT image-management service requests set a bit in the process descriptor's P$State field indicating that the DAT has been changed. This indicates that the DAT will have to be updated with a new DAT image for this process. Changes made to a DAT image aren't effective until they are loaded into the DAT hardware.

TASK NUMBER CONTROL

Task numbers are used to tell the DAT hardware which of the preloaded DAT images it should use. There is an array of service requests that deals with task numbers.

F$SetTsk copies the DAT image from a selected process descriptor into the DAT hardware and clears the ImgChg flag in P$State.

F$ResTsk finds a free task number in the system task table (pointed to by D.Tasks in the system direct page), reserves it, and returns it to the caller. F$AllTsk uses F$ResTsk to reserve a task number and stores that number in the P$Task field in the selected process descriptor.

F$RelTsk returns a task number to the free pool. F$DelTsk uses F$RelTsk to free a task number in the task table, and also clears the task number out of the P$Task field in the process descriptor.

ADDRESS SPACE MANAGEMENT

Two commands are used to find contiguous free blocks in a DAT image. F$FreeHB starts its search from the high end of the DAT image. F$FreeLB starts from the low end. OS-9 allocates stack memory from low memory and system memory from high memory. This measure prevents memory claimed by OS-9 for modules from colliding with any expansion a user might want in his stack memory.

MEMORY MAP MANAGEMENT

The next two service requests are pretty tenuously related, but they both deal exclusively with the system memory map. F$AllRam searches through the memory map for a block of contiguous memory blocks. The DAT makes contiguous memory less important than it is in Level One, but it is nice to be able to find it when you need it.

Those who had early versions of Level Two will remember that there were strange problems with the FORMAT command. The cause of those problems was that FORMAT requires a block of continuous memory; there was no way to request contiguous memory from OS-9, so FORMAT had to take what it could get and return an error message if what it got wasn't contiguous. The CPU has no way of knowing whether memory is contiguous, but a DMA disk controller does. The disk controller doesn't have access to the DAT. It needs to use full extended addresses, and gets upset when a buffer isn't in contiguous memory.

The documentation seems to say that F$AllRAM doesn't do anything but verify the availability of the required number of contiguous blocks. I believe the documentation is in error here. The service request actually allocates the memory if it finds the requested contiguous space, and returns the starting block number in D.

The F$DelRAM service request also works on the system memory map. It marks a range of blocks as not in use.

MISCELLANEOUS SERVICE REQUESTS

Two service requests that don't fit into any class are F$ClrBlk and F$DATLog.

The F$ClrBlk request removes memory from the DAT image of the process who's id is in D.Proc in the system direct page. The blocks removed from the DAT aren't marked as free in the memory block map. F$ClrBlk is very fussy about the starting address of the memory it's instructed to free. The address must be the start of a DAT block; for most systems that means it must be an address like $1000 or $E000. F$ClrBlk will also balk at removing memory being

used by the process's stack from the address space.

F$DATLog is an important call inside OS-9 because the size of DAT blocks can differ from one type of hardware to another. The DAT-plus-offset to logical address conversion depends on the size of the DAT blocks. By putting the conversion in one place, Microware made it easier to write code that will work with any DAT block size. This service request takes a DAT image offset and an offset within the block. Tricky point: the offset doesn't have to be to an address within the block. Any positive offset is fine. On a system with 4K pages, F$DATLog just shifts the DAT image offset to the left by four bits and adds it to the high-order byte of the "block offset."

WORKSHOP 1

cookie monster

Programming can only be properly learned through practice. There is nothing quite so useful to a practicing programmer as a proven program that does something similar to that which he is attempting. This section of the book is full of working programs. Most of them are the husks of useful programs. There is enough here to learn from, but little enough to leave room for you to expand on the ideas.

In some of the programs you will notice what appears to be superfluous code or inconsistency between two sections. It's probably intentional. These programs are set up as illustrations of OS-9 programming techniques. They go out of their way to illustrate a variety of techniques.

The Classic Cookie Program

Cookie is a program with a long (painful) history. It has been written in many forms by college students. The idea is that an unsuspecting user will log onto the computer, and, without doing anything out of the ordinary, be greeted with the message:

I WANT A COOKIE.

This in itself should be enough to send him running for help, but the program doesn't stop there. Well written Cookie programs won't go away. They resist all kill, break, abort, interrupt and whatever other trapdoors a system offers users who find themselves caught in a program. The art of the programmer who created Cookie is pitted against the knowledge of the user. A good Cookie is impossible to kill. The only way to make it go away is to offer it a cookie.

Reply, "Cookie," to any of the program's ravings and it will peacefully go away.

Cookie is an interesting program to study because it must catch signals in order to survive in a hostile environment. If a user could type control-C and have Cookie vanish, there'd be no fun in it at all. It would be possible to remove the interrupt and abort keys with a SETSTAT, but then Cookie wouldn't know that they had been used. With a trap, Cookie can catch signals and choose to ignore them.

The proper way to invoke Cookie is in a login command line or startup file.

I left one easy loophole in this program: end-of-file. You might want to see about closing it.

Note: I have a file called defslist in my DEFS directory. This file is a list of USE command for all the OS9 definition files that came with the system. I routinely run the assembler with over 16K to leave space for the large symbol table that results from all those names, but it prevents me from worrying about where a system name is defined.

The USE /D0/DEFS/DefsList command would normally be suppressed by the IFP1/ENDC that surrounds it. I edited the listing created by the assembler to put it back in.

```
                                nam    Cookie
                                ttl    The classic "Cookie" program
                                IFP1
                                use    /d0/defs/defslist
                                ENDC
     000D             CR        equ    $0D
     000A             LF        equ    $0A
     0011             Type      set    Prgrm+Objct
     0081             Revs      set    ReEnt+1
     0000 87CD0159              mod    ModLen,Name,Type,Revs,Entry,MemSize
   D 0000             SigCode   rmb    1
   D 0001             Indx      rmb    1         Index into STable
   D 0002             STable    rmb    2         Table of response addresses
   D 0004             Table2    rmb    2
   D 0006             Table3    rmb    2
   D 0008             Table4    rmb    2
   D 000A             Table5    rmb    2
   D 000C             Table6    rmb    2
     0006             TableL    set    6         Number of entries in the table
     0050             BufLen    set    80
   D 000E             InStr     rmb    BufLen    Input buffer
   D 005E                       rmb    200       stack
   D 0126             MemSize   equ    .

     000D 436F6F6B    Name      fcs    /Cookie/
     0013 01          Version   fcb    1
          ***********
          *  Messages to the user.  Pointed to from STable
          *
     0014 08          Msg1      fcb    Msg1L
     0015 436F6F6B              fcc    /Cookie/
     001B 0D                    fcb    CR
     0008             Msg1L     equ    *-Msg1
```

```
001C 09         Msg2     fcb   Msg2L
001D 436F6F6B            fcc   /Cookie!/
0024 0D                  fcb   CR
0009            Msg2L    equ   *-Msg2
0025 13         Msg3     fcb   Msg3L
0026 49207761            fcc   /I want a Coookie!/
0037 0D                  fcb   CR
0013            Msg3L    equ   *-Msg3
0038 140A0A     Msg4     fcb   Msg4L,LF,LF
003B 436F6F6B            fcc   /Cookie   N O W/
0049 0A0A                fcb   LF,LF
004B 0D                  fcb   CR
0014            Msg4L    equ   *-Msg4
004C 40         Msg5     fcb   Msg5L
004D 596F7520            fcc   /You can't get rid of me that easily .../
0074 0A                  fcb   LF
0075 2020202A            fcc   /   *** C O O K I E ***/
008B 0D                  fcb   CR
0040            Msg5L    equ   *-Msg5
008C 2D         Msg6     fcb   Msg6L
008D 436F6F6B            fcc   /Cookie!/
0094 0A                  fcb   LF
0095 2020436F            fcc   /  Cookie!/
009E 0A                  fcb   LF
009F 20202020            fcc   /    Cookie!/
00AA 0A                  fcb   LF
00AB 20202020            fcc   /      Cookie!/
00B8 0D                  fcb   CR
002D            Msg6L    equ   *-Msg6
00B9            Sequence
00B9 01                  fcb   1           After 0 comes 1
00BA 02                  fcb   2           after 1 comes 2
00BB 03                  fcb   3           after 2 comes 3
00BC 02                  fcb   2           after 3 comes 2
00BD 03                  fcb   3           after 4 (Sigmessage) comes 3
0005            SeqLen   equ   *-Sequence
00BE            Entry
00BE 0F01                clr   Indx
00C0 0F00                clr   SigCode

00C2 C606                ldb   #TableL
00C4 3142                leay  STable,U
00C6 308DFF4A            leax  Msg1,PCR
00CA            TInit
00CA AFA1                stx   ,Y++
00CC 5A                  decb
00CD 2706                beq   TInitX
00CF A684                lda   ,x          length of string
00D1 3086                leax  A,X         point at next string
00D3 20F5                bra   TInit
00D5            TInitX


     **********
     *  Set interrupt trap
     *
00D5 308D0009            leax  Trap,PCR
     *   intercept local is the same as main storage
00D9 103F09              OS9   F$Icpt
W 00DC 10250073          lbcs  Error
00E0 2003                bra   MainLp      go get a cookie
00E2            Trap
00E2 E7C4                stb   SigCode,U   save the signal code
00E4 3B                  rti               and return to OS-9
```

```
00E5              MainLp
00E5 D601                   ldb   Indx
00E7 8D13                   bsr   GCookie     send a complaint to the termin
00E9 304E                   leax  InStr,U
00EB 108E0050               ldy   #BufLen
00EF 8600                   lda   #0          Standard input Path
00F1 103F8B                 OS9   I$ReadLn    get some data
     * bcs Error
00F4 8D16                   bsr   Food        Is this a Cookie?
00F6 2456                   bcc   Done        If yes; done
00F8 8D37                   bsr   NextMsg
00FA 20E9                   bra   MainLp


     **************
     *  Pick a message and send it
     *  Message number is in B
00FC              GCookie
00FC 58                     lslb              multiply B by two
00FD 3042                   leax  STable,U
00FF AE85                   ldx   B,X         get pointer out of the table
0101 E680                   ldb   ,X+         get string length
0103 4F                     clra
0104 1F02                   tfr   D,Y         length to Y
0106 8601                   lda   #1          standard output
0108 103F8C                 OS9   I$WritLn
010B 39                     rts


     ****************
     *  Compare InStr to "Cookie"
     *  We use the name of the program "Cookie" as
     *  the sample Cookie word.
     *  To compensate for the high order bit in the last "e"
     *  set the high-order bit in the sixth byte of the input string
     *  This will match if the input is Cookie.
     *
010C              Food
010C 304E                   leax  InStr,U
010E 318DFEFB               leay  Name,PCR
0112 A605                   lda   5,X
0114 8A80                   ora   #%10000000 set high bit
0116 A705                   sta   5,X
0118 EC81                   ldd   ,X++
011A 10A3A1                 cmpd  ,Y++
011D 2610                   bne   NotFood
011F EC81                   ldd   ,X++
0121 10A3A1                 cmpd  ,Y++
0124 2609                   bne   NotFood
0126 EC84                   ldd   ,X
0128 10A3A4                 cmpd  ,Y
012B 2602                   bne   NotFood
012D 5F                     clrb              clear carry
012E 39                     rts
012F              NotFood
012F 53                     comb              set carry
0130 39                     rts


     ****************
     *  Choose the successor to the last message.
     *
0131              NextMsg
0131 9600                   lda   SigCode
0133 2612                   bne   SigMsg
0135 9601                   lda   Indx
```

```
0137 8105                  cmpa   #SeqLen    Length of sequence table
0139 2409                  bhs    ToMsg1
013B 308DFF7A              leax   Sequence,PCR
013F A686                  lda    A,X
0141 9701                  sta    Indx
0143 39                    rts
0144           ToMsg1
0144 0F01                  clr    Indx
0146 39                    rts
0147           SigMsg
0147 8604                  lda    #TableL-2  Signal Message
0149 9701                  sta    Indx
014B 0F00                  clr    SigCode
014D 39                    rts
014E           Done
014E C605                  ldb    #TableL-1  last message is thankyou
0150 8DAA                  bsr    GCookie
0152 5F                    clrb              clear carry
0153           Error
0153 103F06                OS9    F$Exit
0156 370A5F                EMOD
0159           ModLen      equ    *

00000 error(s)
00001 warning(s)
$0159 00345 program bytes generated
$0126 00294 data bytes allocated
$241F 09247 bytes used for symbols
```

A Daemon

According to my New World Dictionary, a daemon is ". . . a guardian spirit." Every computer needs a guardian spirit. The Daemon that follows will sit in your system, running programs for you. He can be told to run a program every so many seconds, or at set times and dates.

If you want to be reminded to dump your hard disk on the first of every month (too seldom), tell the Daemon to:

echo BACK UP /H0 NOW!! >/term

at the time:

YY/MM/DD HH:MM:SS
***/*/1 17:01:01**

That will tell him to give you the message at five in the afternoon on the first of each month. The asterisks are wild cards that tell the deamon that any value of that variable is OK.

The basic operation of the Daemon isn't too interesting. He sleeps for some length of time (I chose 10 seconds), then wakes up and sees if there is anything he should do. He does his business and goes to sleep again. All the commands the Daemon executes are simple shell commands. He forks a shell to interpret each one.

The interesting part of the Daemon is his communication with

the rest of the world. There needs to be a way to send instructions to him. Signals are a good way to joggle a process's elbow, but a signal doesn't have much information content. A disk file can contain plenty of information, but it wouldn't be good for the Daemon to open a disk file every 10 seconds. Someone might want to remove the disk with the Daemon's communication file on it. Pipes are a good tool for inter-process communication, but they can only run between closely related processes. The process talking to the Daemon might be a very distant relative. The solution is a shared data module.

A shared data module can be tricky. If more than one process might want to update it at a given time, you have to build a locking system to prevent them from wiping out one another's updates. In this case we have to handle any number of processes that simultaneously attempt to send requests to the Daemon; so we've got problems.

The solution is to limit communications through the data module to a bandwidth almost as narrow as signals offer, and pull signals and communication files into the act. The process id of the Daemon is stored in a data module that has a known name. When a process wants to send a request to the Daemon it opens the Daemon's command file and concatenates the new command to it, then closes it. It is important to close the file immediately, because the file has the non-sharable attribute. The daemon cannot get at his own file if another process has it open. Next, the process links to the data module and gets the process id in it. It also sets a flag in the data module. If other processes also set the flag, no problem, it's either set or not. The process then sends a "wake-up" signal to the Daemon; this terminates his sleep. The first thing the Daemon does is check the flag in the data module. If it's set, he clears it and reads the command file.

The problem of multiple writers is shoved off on OS-9. RBFMan can deal with contention for a non-sharable file. The problem with frequent accesses to a communication file is eliminated because the Daemon will only access the file when he knows something is there for him. That won't cause any trouble because the program that sent the request to the Daemon already forced the user to load the correct disk.

The Daemon is a single program (two files), but a usable system includes three other programs and the data module. The data module is called Daemon.com, you'll find it exceptionally dull reading. There is a program called Then (Some people would prefer At — go ahead and change the name), which sends new commands to the Daemon. PeekDaemon formats and displays the commands in the Daemon's command file.

Daemon MUST have Daemon.com in memory. The best way to do this is to merge both modules into one file in your execution directory. The following script will do the trick:

```
OS9:shell
OS9:chd /d0/cmds
OS9:merge Daemon Daemon.com >temp
OS9:del Daemon
OS9:rename temp Daemon
OS9:attr Daemon pe e
OS9:<eof>
```

```
*********Daemon.h ************
#define ECMDLEN 150
struct EventRecord
 {
    int intv;
    struct sgtbuf settime;
    char cmdLine[ECMDLEN];
 }

#define ifilen "/SYS/TimeT"
#define Com_datamod "Daemon.com"
```

```
*********************** Daemon.c *******************
#define LEVEL2      /* if running under OS-9 Level Two */
#include <module.h>
#include <stdio.h>
#include <time.h>
#include "Daemon.h"
#define MAXEVENTS 50
#define tick 10*tps     /* ten seconds */
#define ifilen "/SYS/TimeT"
#define TRUE 1
#define FALSE 0
#define DataType '\x40'
#define DataLang '\0'

struct Events
 {
    int interval;
    long lastHit;
    struct sgtbuf Set_Time;
    char *CmdLine;
 };

struct Events Event_Table[MAXEVENTS];
int Event_Count=0;
static int sig=0;
main()
 {
    int intf();

    InitEvents(Event_Table, &Event_Count);
    intercept(intf);
    GetDMod();


    while(sig == 0)
       {
          UpdEvents(Event_Table, &Event_Count);
          DoEvents(Event_Table, Event_Count);
          tsleep(tick);
       }
    DropMod();
```

```
    exit(0);
}

InitEvents(Table, Ct)
   struct Events Table[];
   int *Ct;
   {
      char *defdrive();
      char NameInitFile[50];

      struct EventRecord FileEntry;
      struct sgtbuf tbuffer;

      FILE *IFile;
      char *malloc();
      long toseconds(), thisSec;

      strcpy(NameInitFile,defdrive());
      strcat(NameInitFile,ifilen);
      if ((IFile = fopen(NameInitFile,"r")) == NULL)
         {
            fprintf(stderr,"Init file can't be opened. Error %d\n", errno);
            exit(1);
         }
      getime(&tbuffer);
      thisSec = toseconds(&tbuffer);
      while (fread(&FileEntry, sizeof FileEntry, 1, IFile) != NULL)
         {
            Table[*Ct].CmdLine = malloc(strlen(FileEntry.cmdLine) +  1);
            strcpy(Table[*Ct].CmdLine,FileEntry.cmdLine);

            Table[*Ct].interval = FileEntry.intv;
            Table[*Ct].lastHit = thisSec;
            _strass(&Table[*Ct].Set_Time, &FileEntry.settime,
                    sizeof FileEntry.settime);
            if((*Ct)++ >= MAXEVENTS)
               break;
         }
      fclose(IFile);
      return;
   }


static char *dMod = Com_datamod;
static mod_exec *modlink(), *mod_ptr=-1;
static char *flagptr;
static int *TaskPtr;

 GetDMod()
   {
      if((mod_ptr = modlink(dMod, DataType, DataLang)) == -1)
         {
            fprintf(stderr, "Can't link to %s.", dMod);
            exit(1);
         }
      flagptr = mod_ptr + mod_ptr->m_exec;
      TaskPtr = flagptr + 1;
      *TaskPtr = getpid();
      return;
   }

 DropMod()
   {
      if(mod_ptr != -1)
         munlink(mod_ptr);
      return;
```

```
   }

UpdEvents(Table, Ct)
   struct Events Table[];
   int *Ct;
   {
      while(*flagptr != 0)
         {
            *flagptr = 0;
            *Ct = 0;
            fprintf(stderr,"Daemon update request\n");
            InitEvents(Table,Ct);
         }
      return;
   }

DoEvents(Table,Ct)
   struct Events Table[];
   int Ct;
   {
      struct sgtbuf tbuffer;
      register int i;
      long now;

      getime(&tbuffer);
      now = tosecond(&tbuffer);
      for(i=0;i<Ct;i++)
         {
            if(Table[i].interval >= 0)
               {
                  if (Table[i].interval <= (now - Table[i].lastHit))
                     {
                        system(Table[i].CmdLine);
                        Table[i].lastHit = now;
                     }
               }
            else
               if(match(&tbuffer,&Table[i].Set_Time,&Table[i].lastHit))
                  system(Table[i].CmdLine);
         }
      return;
   }

long toseconds(b)
 struct sgtbuf *b;
 {
    return(b->t_second + 60*(b->t_minute + (60 * b->t_hour)));
 }

match(t,pat,lastHit)
 struct sgtbuf *t, *pat;
 long *lastHit;
 {
    if((pat->t_year > 0) && (pat->t_year != t->t_year))
       ;
    else if((pat->t_month > 0) && (pat->t_month != t->t_month))
       ;
    else if((pat->t_day > 0) && (pat->t_day != t->t_day))
       ;
    else if((pat->t_hour > 0) && (pat->t_hour != t->t_hour))
       ;
    else if((pat->t_minute > 0) && (pat->t_minute  != t->t_minute))
       ;
    else if((pat->t_second > 0) && (pat->t_second - 5 > t->t_second ||
                                    pat->t_second + 4 < t->t_second))
       ;
```

```
    else
       {
          if(*lastHit > 0)
             return(FALSE);
          else
             {
                *lastHit = TRUE;
                return(TRUE);
             }
       }

    *lastHit = FALSE;

    return(FALSE);
}

intf(signal)
   int signal;
   {
      if(signal > 1)
         sig = signal;
      return;
   }
```

****************** Daemon.Com ********************

```
                                 nam   Daemon.Com
                                 ttl   Data Module for Daemon
                                 IFP1
                                 ENDC
 0040                  Type      set   Data
 0081                  Revs      set   ReEnt+1
 0000 87CD001E                   mod   Size,Name,Type,Revs,Start,0
 000D 4461656D         Name      fcs   /Daemon.com/
 0017 01                         fcb   1           version
 0018                  Start
 0018 00               Flag      fcb   0
 0019 0000             DTask     fdb   0
 001B 930A0A                     EMOD
 001E                  Size      equ   *
```

********************** Then.c ********************

```
#include <stdio.h>
#include <module.h>
#include <ctype.h>
#include <time.h>
#include <modes.h>
#include <errno.h>
#include <signal.h>
#include "Daemon.h"

#define DataType '\x40'
#define DataLang '\0'

static char *usage[] =
 {
    "The format of the Then command is:",
    "     Then <shell command line>",
    "If the command line must contain shell operators:",
    "(;><&)",
    "don't put anything on the command line.  Then will prompt",
    "for a command.",
    "",
    "Then will always prompt for directions on when to execute the",
    "shell command."
 };
```

```

static struct EventRecord FileEntry;

main(argc,argv)
 int argc;
 char **argv;
 {
    int i;
    char c;
    FILE *IFile, *OpenTimer();

    argv++;
    if(argc == 2 && **argv == '?')
       {
          directions();
          exit(0);
       }


    setbuf(stdin, NULL); /* unbuffered input */
    setbuf(stdout, NULL); /* unbuffered output */

    /*-------------------------------------------*
     *         Build the Command Line             *
     *-------------------------------------------*/
    FileEntry.cmdLine[0] = '\0';
    if(argc < 2)
       {
          printf("SHELL CMD: ");
          fgets(FileEntry.cmdLine,ECMDLEN,stdin);
          printf("\n");
       }
    else
       for(i=2; i<=argc; i++)
          {
             strcat(FileEntry.cmdLine,*argv++);
             strcat(FileEntry.cmdLine," ");
          }

   /*-------------------------------------------*
    *   Select set time or set interval          *
    *-------------------------------------------*/
    do
       {
          printf("Execute the command at a set time? (Y,N): ");
          c = toupper(getchar());
       }  while (c != 'Y' && c != 'N');
    printf("\n");

    FileEntry.intv = -1;                 /* Initialize FileEntry */
    FileEntry.settime.t_year = -1;

    if(c == 'Y')
       Time();      /* Set time for execution */
    else
       Interval();  /* Set interval for execution */

    IFile = OpenTimer();

    fwrite(&FileEntry, sizeof FileEntry, 1, IFile);
    flagDaemon();
    fclose(IFile); /* release the timer file */
    exit(0);
 }

 Time()
```

```
    {

       int i;

       printf("Enter time  YY/MM/DD  HH:MM:SS\n");
       printf("Use * as a wild card\n");
       printf("YY/MM/DD HH:MM:SS\n");

       FileEntry.settime.t_year = getnum();
       putchar('/');
       FileEntry.settime.t_month = getnum();
       putchar('/');
       FileEntry.settime.t_day = getnum();
       putchar(' ');
       FileEntry.settime.t_hour = getnum();
       putchar(':');
       FileEntry.settime.t_minute = getnum();
       putchar(':');
       FileEntry.settime.t_second = getnum();
       printf("\n");

       return;
    }

 getnum()
    {
       char c1, c2;

       do
         c1 = getchar();
       while (!isdigit(c1) && (c1 != '\n') && (c1 != '*'));

       if(c1 == '*' || c1 == '\n')
          return(0);

       do
          c2 = getchar();
       while (! isdigit(c2) && c2 != '\n');

       if(c2 =='\n')
          return(c1 - '0');
       else
          return((c2 - '0') + ((c1 - '0') * 10));
    }

 Interval()
    {
       printf("Interval:\n");
       printf("HH:MM:SS\n");
       FileEntry.intv = getnum()*60*60;
       putchar(':');
       FileEntry.intv += getnum()*60;
       putchar(':');
       FileEntry.intv += getnum();
       printf("\n");
       return;
    }

static char modname[] = Com_datamod;

 flagDaemon()
    {
       mod_exec *modlink();
       mod_exec *mod_ptr;
       char *flagptr;
       int *DaemonTask;
```

```
      if ((mod_ptr = modlink(modname, DataType, DataLang)) == -1)
         {
            fprintf(stderr, "Can't link to %s. Error %d\n", modname, errno);
            exit(1);
         }
      flagptr = mod_ptr + mod_ptr->m_exec;
      DaemonTask = flagptr + 1;
      (*flagptr)++;

      fprintf(stderr,"Signaling task %d\n",*DaemonTask);
      kill(*DaemonTask,SIGWAKE); /* joggle the Daemon's elbow */

      munlink(mod_ptr);

      return;
   }


FILE *OpenTimer()
 {
   char NameInitFile[50];
   char *defdrive();
   char new;
   FILE *IFile;

   strcpy(NameInitFile,defdrive());  /* Build timer file name */
   strcat(NameInitFile,ifilen);

   new = access(NameInitFile,0);
   if((IFile = fopen(NameInitFile,"a")) == NULL)  /* Open timer file */
      {
         if(errno == E_BPNAM)
            fprintf(stderr,"%s %s.\n",
                    "A disk containing the SYS directory must be in",
                    defdrive());
         fprintf(stderr, "Init file can't be opened. Error %d\n",
                  errno);
         exit(1);
      }
   if(new)
      chmod(NameInitFile,S_IREAD+S_IWRITE+S_IOREAD+S_IOWRITE+S_ISHARE);
   return(IFile);
 }

  directions()
   {
      register int i;

      for(i=0; i < sizeof usage / sizeof (char *); i++)
         puts(usage[i]);

      return;
   }

************** PeekDaemon.c ******************
#include <stdio.h>
#include <time.h>
#include "Daemon.h"

static struct EventRecord FileEntry;

main()
 {
 FILE *IFile;
```

```
char NameInitFile[50], *defdrive();
char DateStr[20];
char *nstr();

strcpy(NameInitFile,defdrive());
strcat(NameInitFile,ifilen);
if((IFile = fopen(NameInitFile,"r")) == NULL)
   {
      fprintf(stderr, "Init file can't be opened.  Error %d\n",errno);
      exit(1);
   }

while (fread(&FileEntry, sizeof FileEntry, 1, IFile) != NULL)
   {
      if(FileEntry.intv == -1)
         {
            printf("%s/%s/%s %s:%s:%s",
                  nstr(FileEntry.settime.t_year),
                  nstr(FileEntry.settime.t_month),
                  nstr(FileEntry.settime.t_day),
                  nstr(FileEntry.settime.t_hour),
                  nstr(FileEntry.settime.t_minute),
                  nstr(FileEntry.settime.t_second));
            reset();
         }
      else
         printf("%d",FileEntry.intv);

      printf(" ==> %s\n",FileEntry.cmdLine);
   }
exit(0);
}

static char wildcard[2] = {'*','\0'};
static char s[6][3];
static int sptr=0;

char *nstr(n)
 int n;
 {
    if(n <= 0)
       return(wildcard);

    sprintf(s[sptr],"%d",n);

    return(s[sptr++]);
 }

reset()
 {
    sptr = 0;
 }
```

WORKSHOP 2

a notepad

The Daemon uses a data module to store a few important bits of information. Larger data modules also have their uses. One way to think of them is as global storage that persists even between programs. Fortran programmers might find the metaphor of common storage useful. A simple use of this kind of global storage is notepad storage. A Notepad can be kept on disk, but then you'd have to load the disk containing the notepad every time you wanted a look at it. I wouldn't use a notepad if it involved switching disks. For those of us with extra large disks this is no problem. For others . . .

A notepad kept in a data module will have to be small, but that's in the nature of notepads. If you want to record large amounts of data, use a file.

This system of programs includes a set of BASIC09 modules. Note is a master program that invokes most of the others. NoteS is a way to see the contents of the Notepad without going through the menus in Note.

GetNote is a set of assembly language subroutines meant to be called from BASIC09. They link to the Notebook data module and unlink it; they also copy a block of data to and from the data module. These modules are interesting both as examples of the use of a data module, and as assembler subroutines for BASIC09. In the first one, I test the length of a parameter. That's an example of something thrown in just because it seemed to need demonstrating.

```
PROCEDURE note
 0000      (*
 0003      (* Driver program for notebook maintenance
 002D      (*
 0030      DIM workstring:STRING[500] \(* Data copied from the notebook *)
 005F      DIM module:INTEGER \(* The address of the notebook data module *)
 0093      DIM Selection:STRING[1]
 009F      (*
 00A2      (* Prompt for a command
 00B9      (*
 00BC      REPEAT
 00BE        REM You might want to insert code to clear your screen here
 00F8        PRINT  \ PRINT  \ PRINT
 00FE        PRINT TAB(10); "NotePad Menu"
 0112        PRINT  \ PRINT "A    Load NotePad from disk (file NotePad)"
 0142        PRINT "B    Save NotePad to disk (file NotePad)"
 016E        PRINT "C    Edit NotePad"
 0183        PRINT  \ PRINT TAB(10); "Selection: "
 0198        INPUT Selection
           UNTIL Selection >= "A" AND Selection <= "C" OR Selection >= "a" AND
                 Selection<="c"
 01C1      (*
 01C4      (* Correct Selection to upper case
 01E6      (*
 01E9      IF Selection>"Z" THEN
 01F6        Selection=CHR$(ASC(Selection)-32)
 0203      ENDIF
 0205      RUN GNoteB(module) \(* Link to the notebook data module *)
 0235      (*
 0238      (* run a procedure to execute the selection
 0263      (*
 0266      IF Selection="A" THEN
 0273        RUN LoadNote(workstring,module)
 0282      ELSE IF Selection="B" THEN
 0292          RUN SaveNote(workstring,module)
 02A1        ELSE IF Selection="C" THEN
 02B1            RUN EditNote(workstring,module)
 02C0          ENDIF
 02C2        ENDIF
 02C4      ENDIF
 02C6      RUN DNote(module) \(* Unlink the notebook data module *)
 02F5      END

PROCEDURE LoadNote
 0000      PARAM S:STRING[500]
 000C      PARAM Module:INTEGER
 0013      DIM i,j:INTEGER
 001E      DIM NoteFile:INTEGER
 0025      ON ERROR GOTO 100
 002B      OPEN #NoteFile,"NotePad":READ
 003D      GET #NoteFile,S
 0047      RUN PNote(Module,S)
 0056      CLOSE #NoteFile
 005C      END
 005E 100  ON ERROR
 0064      S=CHR$(13)
 006C      END

PROCEDURE SaveNote
 0000      PARAM S:STRING[500]
 000C      PARAM Module:INTEGER
 0013      DIM NoteFile:INTEGER
 001A      CREATE #NoteFile,"NotePad":WRITE
 002C      RUN GNote(Module,S)
```

```
 003B      PUT #NoteFile,S
 0045      CLOSE #NoteFile
 004B      END

PROCEDURE EditNote
 0000      (*
 0003      (* EditNote is a very simple editor for the notes in
 0037      (* the notebook.
 0047      (* It only can list single lines, and add and delete lines
 0081      (*
 0084      PARAM S:STRING[500]
 0090      PARAM Module:INTEGER
 0097      DIM start,last:INTEGER
 00A2      DIM Selection:STRING[1]
 00AE      DIM Line:STRING[80] \(* Input buffer for new lines *)
 00DA      DIM temp:STRING[500] \(* Work space for insert-delete operations *)
 0113      RUN GNote(Module,S) \(* Get notes from Notebook in S *)
 0144      GOSUB 100 \(* Give Help Message *)
 015F      last=0
 0166      GOSUB 90 \(* Parse out a line *)
 0180      PRINT
 0182      REPEAT
 0184        PRINT MID$(S,start,last-start); " ->";  \(* Print line, prompt  *)
 01B5        INPUT Selection
 01BA        (*
 01BD        (* Ensure that Selection is upper case
 01E3        (*
 01E6        IF Selection>"Z" THEN
 01F3          Selection=CHR$(ASC(Selection)-32)
 0200        ENDIF
 0202        (*
 0205        (* Execute the command
 021B        (*
 021E        IF LEN(Selection)=0 THEN  \(* Input was a return *)
 0244          GOSUB 90 \(* Parse out next line *)
 0261        ELSE IF Selection="A" THEN  \(* Add a new line *)
 0286            INPUT ": ",Line
 0290            IF LEN(Line)+LEN(S)>500 THEN
 02A3              PRINT "Notepad overflow.  Addition rejected"
 02CB            ELSE
 02CF              temp=LEFT$(S,start-1)
 02DE              temp=temp+Line+CHR$(13)
 02EE              temp=temp+RIGHT$(S,LEN(S)+1-start)
 0306              S=temp
 030E              last=start+LEN(Line)
 031B            ENDIF
 031D          ELSE IF Selection="D" THEN  \(* Delete the current line *)
 034B              temp=LEFT$(S,start-1)+RIGHT$(S,LEN(S)-last)
 0367              S=temp
 036F            ELSE IF Selection="H" OR Selection="?" THEN
 0387                GOSUB 100
 038B              ELSE IF Selection="Q" THEN  \(* Quit *)
 03A6                ELSE IF Selection="-" THEN  \(* Back up a line *)
 03CB                    IF start=1 THEN
 03D7                      PRINT "*****  Note can't be backed up past first note."
 040A                    ELSE
 040E                      last=start-1
 0419                      IF last<1 THEN
 0425                        last=1
 042C                      ENDIF
 042E                      start=last-1
 0439                      WHILE start>=1 AND MID$(S,start,1)<>CHR$(13) DO
 0453                        start=start-1
 045E                      ENDWHILE
 0462                      IF start<>last THEN
 046F                        start=start+1
```

```
047A                    ELSE
047E                      GOSUB 90
0482                    ENDIF
0484                  ENDIF
0486                ENDIF
0488              ENDIF
048A            ENDIF
048C          ENDIF
048E        ENDIF
0490      ENDIF
0492    UNTIL Selection="Q"
049E    RUN PNote(Module,S) \(* Copy data back into notebook data module *)
04DB    END
04DD 90 REM Parse out a line
04F3    IF last+1<LEN(S) THEN
0504      start=last+1
050F    ELSE
0513      start=1
051A    ENDIF
051C    last=start
0524    WHILE MID$(S,last,1)<>CHR$(13) AND last<LEN(S) DO
0540      last=last+1
054B    ENDWHILE
054F    RETURN
0551 100 REM Help Message
0563    PRINT "Return to see the next line."
0583    PRINT "D      to delete the current line."
05A9    PRINT "A      to add a line before the current line."
05DA    PRINT "-      to see the previous line."
05FE    PRINT "Q      to quit editing the Notepad."
0625    PRINT "H      to see this help message."
0649    RETURN

PROCEDURE notes
0000    (*
0003    (* Just list the notes in the NoteBook
0029    (* Don't prompt for options
0044    (*
0047    DIM start,last:INTEGER
0052    DIM temp:STRING[500] \(* storage for the data from NoteBook *)
0086    DIM Module:INTEGER \(* The address of the NoteBook data module *)
00BA    RUN GNoteB(Module) \(* Link to the data module *)
00E1    RUN GNote(Module,temp) \(* Copy notebook data to temp *)
0110    last=0
0117    GOSUB 90 \(* Parse out a line *)
0131    PRINT
0133    REPEAT
0135      PRINT MID$(temp,start,last-start) \(* Print one line from the notebook
016B      GOSUB 90 \(* Parse out a line *)
0185    UNTIL start=1
0190    RUN DNote(Module) \(* Unlink the notebook data module *)
01BF    END
01C1 90 REM Parse out a line
01D7    IF last+1<LEN(temp) THEN
01E8      start=last+1
01F3    ELSE
01F7      start=1
01FE    ENDIF
0200    last=start
0208    WHILE MID$(temp,last,1)<>CHR$(13) AND last<LEN(temp) DO
0224      last=last+1
022F    ENDWHILE
0233    RETURN

************************GetNote   **********************
00001                          nam   GetNote
```

```
                        ttl   Get Information from NoteBook for Basic09
                        IFP1
                        use   /D0/DEFS/defslist
                        ENDC
  0021          type    set   Objct+Sbrtn
  0081          Revs    set   ReEnt+1
  0000 87CD0043         mod   GNoteBS,Name1,Type,Revs,GNoteB,0
       *******
       * Parameter area
       *
D 0000                  org   0
D 0000          RetAddr rmb   2            return address
D 0002          ParmCt  rmb   2            number of parameters
D 0004          V1      rmb   2            address of module address
D 0006          V1L     rmb   2            length of V1
D 0008          V2      rmb   2
D 000A          V2L     rmb   2
D 000C          V3      rmb   2
D 000E          V3L     rmb   2
D 0010          V4      rmb   2
D 0012          V4L     rmb   2
D 0014          V5      rmb   2
D 0016          V5L     rmb   2
       * End of parameter area
  000D 474E6F74 Name1   fcs   /GNoteB/
  0013 4E6F7465 NoteBook fcs  /NoteBook/
  001B 01               fcb   1
       ************
       *  Link to NoteBook data module.  Return the address of
       *  the module header to the caller
       *
  001C          GNoteB
  001C EC62             ldd   ParmCt,S     get parameter count
  001E 10830001         cmpd  #1           expect one
  0022 2618             bne   ParamErr
  0024 EC66             ldd   V1L,S
  0026 10830002         cmpd  #2           expect two
  002A 2610             bne   ParamErr
       ********
       *  Link to NoteBook
  002C 8640             lda   #Data
  002E 308DFFE1         leax  NoteBook,PCR
  0032 103F00           OS9   F$Link
  0035 2508             bcs   Error
  0037 EFF804           stu   [V1,S]
  003A 5F               clrb
  003B 39               rts                return
  003C          ParamErr
  003C C638             ldb   #E$Param
  003E 43               coma               set carry
  003F          Error
  003F 39               rts
  0040 B0F864           EMOD
  0043          GNoteBS equ   *
```

```
       *************
       *  Unlink NoteBook data module
       *
  0000 87CD0034         mod   DNoteS,Name2,Type,Revs,DNote,0
  000D 444E6F74 Name2   fcs   /DNote/
```

```
0012 4E6F7465   NoteBk2  fcs   /NoteBook/
001A 01                  fcb   1
001B            DNote
001B EC62                ldd   ParmCt,S    get parameter count
001D 10830001            cmpd  #1          expect one
0021 260A                bne   ParmErr2
     ********
     *  UnLink NoteBook
     *
0023 EEF804              ldu   [V1,S]
0026 103F02              OS9   F$UnLink
0029 2505                bcs   Error2
002B 5F                  clrb
002C 39                  rts
002D            ParmErr2
002D C638                ldb   #E$Param
002F 43                  coma              set carry
0030            Error2
0030 39                  rts
0031 786B44              EMOD
0034            DNoteS   equu  *

     *************
     *  store info in NoteBook
     *  run WSet(ModPtr,Data)
     *  Module pointer, 500 bytes of data
     *
0000 87CD0041            mod   PNoteS,Name3,Type,Revs,PNote,0
000D 504E6F74   Name3    fcs   /PNote/
0012 01                  fcb   1
0013            PNote
0013 EC62                ldd   ParmCt,S    get parameter count
0015 10830002            cmpd  #2          expect two
0019 2614                bne   ParmErr3
001B 10AEF804            ldy   [V1,S]      address of module
001F EC29                ldd   9,Y         data offset
0021 31AB                leay  D,Y         Y points at data in NoteBook
0023 AE68                ldx   V2,S
0025 C6FF                ldb   #255        max move size
0027 8D0A                bsr   Move        move X to Y length B
0029 C6F5                ldb   #500-255    remaining length
002B 8D06                bsr   Move        continue move
002D            Exit
002D 5F                  clrb
002E 39                  rts
002F            ParmErr3
002F C638                ldb   #E$Param
0031 43                  coma              set carry
0032            Error3
0032 39                  rts
0033            Move
0033 5D                  tstb
0034 2707                beq   MoveX
0036            MoveLoop
0036 A680                lda   ,X+
0038 A7A0                sta   ,Y+
003A 5A                  decb
003B 26F9                bne   MoveLoop
003D            MoveX
003D 39                  rts
003E 713917              EMOD
0041            PNoteS   equ   *

     *************
     *  get info from NoteBook
     *  run GNote(ModPtr,Data)
```

```
        *  (Module pointer, 500 bytes of unformatted data)
        *
0000 87CD0041                mod     GNoteS,Name4,Type,Revs,GNote,0
000D 474E6F74   Name4        fcs     /GNote/
0012 01                      fcb     1
0013            GNote
0013 EC62                    ldd     ParmCt,S     get parameter count
0015 10830002                cmpd    #2           expect Two
0019 2614                    bne     ParmErr4
001B 10AEF804                ldy     [V1,S]       address of module
001F EC29                    ldd     9,Y          data offset
0021 31AB                    leay    D,Y          point at data in NoteBook
0023 AE68                    ldx     V2,S         point at data from Basic09
0025 C6FF                    ldb     #255         Max length to move
0027 8D0A                    bsr     Move.2       move [Y] to [X] for B
0029 C6F5                    ldb     #500-255     remaining length
002B 8D06                    bsr     Move.2
002D            Exit.2
002D 5F                      clrb
002E 39                      rts
002F            ParmErr4
002F C638                    ldb     #E$Param
0031 43                      coma                 set carry
0032            Error4
0032 39                      rts
0033            Move.2
0033 5D                      tstb
0034 2707                    beq     MoveX.2
0036            MovLoop2
0036 A6A0                    lda     ,Y+
0038 A780                    sta     ,X+
003A 5A                      decb
003B 26F9                    bne     MovLoop2
003D            MoveX.2
003D 39                      rts
003E 9AB744                  EMOD
0041            GNoteS       equ     *

00000 error(s)
00000 warning(s)
$00F9 00249 program bytes generated
$0018 00024 data bytes allocated
$2410 09232 bytes used for symbols
```

```
********************** NoteBook data module ****************

                             nam     NoteBook
                             ttl     Data Module for Note system
                             IFP1
                             use     /d0/DEFS/defslist
                             ENDC
0040            type         set     Data
0081            Revs         set     ReEnt+1
0000 87CD020C                mod     MSize,Name,Type,Revs,DStart,DSize
000D 4E6F7465   Name         fcs     /NoteBook/
0015            DStart
0015 0DFF0000                fcb     13,$FF,0,0,0,0,0,0,0,0 ten bytes 1
001F 00000000                fcb     0,0,0,0,0,0,0,0,0,0 ten bytes 2
0029 00000000                fcb     0,0,0,0,0,0,0,0,0,0 ten bytes 3
0033 00000000                fcb     0,0,0,0,0,0,0,0,0,0 ten bytes 4
003D 00000000                fcb     0,0,0,0,0,0,0,0,0,0 ten bytes 5
0047 00000000                fcb     0,0,0,0,0,0,0,0,0,0 ten bytes 6
0051 00000000                fcb     0,0,0,0,0,0,0,0,0,0 ten bytes 7
005B 00000000                fcb     0,0,0,0,0,0,0,0,0,0 ten bytes 8
0065 00000000                fcb     0,0,0,0,0,0,0,0,0,0 ten bytes 9
```

```
006F 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 10
0079 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 11
0083 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 2
008D 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 3
0097 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 4
00A1 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 5
00AB 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 6
00B5 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 7
00BF 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 8
00C9 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 9
00D3 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 20 (200 byt
00DD 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 1
00E7 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 2
00F1 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 3
00FB 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 4
0105 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 5
010F 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 6
0119 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 7
0123 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 8
012D 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 9
0137 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 30
0141 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 1
014B 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 2
0155 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 3
015F 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 4
0169 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 5
0173 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 6
017D 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 7
0187 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 8
0191 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 9
019B 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 40 (400 byt
01A5 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 1
01AF 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 2
01B9 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 3
01C3 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 4
01CD 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 5
01D7 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 6
01E1 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 7
01EB 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 8
01F5 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 9
01FF 00000000          fcb   0,0,0,0,0,0,0,0,0,0 ten bytes 50 (500 byt
01F4          DSize    equ   *-DStart
0209 1C79BD            EMOD
020C          MSize    equ   *

00000 error(s)
00000 warning(s)
$020C 00524 program bytes generated
$0000 00000 data bytes allocated
$21C7 08647 bytes used for symbols
```

'more'

A PROGRAM TO PAGINATE TO THE TERMINAL

If you have pause turned off (using TMODE), listings will fly by on your terminal so fast you won't be able to do more than appreciate their length. If you turn pause on, you can see one page at a time, but there isn't an easy way to move down a line at a time. Moving through a file 24 lines at a crack leaves you looking at each fresh path with no context from the previous page.

The next program, More, isn't a large improvement over the pagination that SCFMan provides, but it has some nice features. Most important, it demonstrates an application of pipes. You'll also find examples of the use of FMem, FFork, F$Icpt and F$Send.

There is one problem that this program wrestles with without much success. More should be able to act on any shell command. The built-in shell commands are problems, but a more annoying problem is that the shell insists on seeing various special characters — '#' and '!' for example. There is no way to prevent the shell from acting on these characters, so I gave up and designed More to be able to accept a shell command line from standard input. You'll notice that it checks the parameter area for a command line. If there is only a <return> character in the parameter area, More prompts for a command line. This gets around the problem, but it doesn't solve it.

For a simple command like LIST TEMP, use More like this:

OS9: more list temp

At the bottom of each page you will get a "More:" prompt. Here you have four options:

<return>	**next page**
<space>	**next line**
<line feed>	**list the rest of the file without a pause**
<tab>	**quit**

Quiting was a problem. It's easy to exit, but making certain the programs down the pipe also died was harder. The programs would get an error on their next write to the pipe, but what if they didn't write again. One way to handle the problem would be to just terminate display of the input. More could go on to read all the input from the pipe and throw it away. I chose to remember the process id of the shell that was forked earlier. When More terminates, with a <tab> command or a signal, it sends a kill signal to its child.

******************** **More** ********************

```
                                  nam    More
                                  ttl    Paginate input for a terminal
                                  IFP1
                                  ENDC
   0011               Type        set    Prgrm+Objct
   0081               Revs        set    ReEnt+1
   0000 87CD01C9                  mod    MSize,Name,Type,Revs,Enter,DSize
        *****************
        *  Variables
        *
D  0000               PPageLen    rmb    1             length of standard out page
D  0001               CPageLen    rmb    1             length to copy
D  0002               Pause       rmb    1             init. pause state
D  0003               PipeNo      rmb    1             path number of pipe
D  0004               StdOSave    rmb    1             path number of std. out copy
D  0005               ChildNo     rmb    1             task number of child
D  0006               Signal      rmb    1             signal received
D  0007                           rmb    350           Stack space
D  0165               DSize       equ    .
   000D 4D6F72E5      Name        fcs    /More/
   0011 01            Version     fcb    1
   0012 4D6F7265      Prompt      fcc    /More: /
   0006               PromptL     equ    *-Prompt
   0018 2F504950      Pipe        fcs    "/PIPE"
   001D 5348454C      SHELL       fcs    "SHELL"
   0022               Enter
   0022 3436                      pshs   D,X,Y
   0024 0F06                      clr    Signal        no signal received
        ***************
        *  First get page length from the standard output path
        *  and set pause off in that path.
        *
   0026 8601                      lda    #1            standard output path number
   0028 C600                      ldb    #SS.Opt       getstat code
   002A 32E8E0                    leas   -32,S         make space on the stack
   002D 30E4                      leax   ,S            point X at the temp storage on
   002F 103F8D                    OS9    I$GetStt
```

```
0032 10250166            lbcs   Error
0036 A608                lda    PD.PAG-PD.OPT,X get page length
0038 9700                sta    PPageLen
003A A607                lda    PD.PAU-PD.OPT,X get page-pause byte
003C 9702                sta    Pause          save it
003E 6F07                clr    PD.PAU-PD.OPT,X set no page-pause
0040 8601                lda    #1             standard output
     * B is still SS.Opt from GetStat
     * X still points to the temp storage
0042 103F8E              OS9    I$SetStt       set pause off
0045 10250153            lbcs   Error
0049 32E820              leas   32,S           clear temp space off the stack
     *************
     *  See if a command line was given
     *
004C 3536                puls   D,X,Y          get initial register contents
004E 3406                pshs   D              save param area length again
0050 A684                lda    ,X             get first byte in the param. a
0052 810D                cmpa   #$0D           <CR>?
0054 2620                bne    GotParam       no; there's something in the p
     ************
     *  Get a command from standard input
     *
0056 3262                leas   2,S            clear parameter area length of
0058 CC0000              ldd    #0             request memory size
005B 103F07              OS9    F$Mem          get mem size in D
005E C30100              addd   #256           add space for command buffer
0061 103F07              OS9    F$Mem          get the buffer
0064 10250134            lbcs   Error
0068 1700BF              lbsr   WPrompt
006B 8600                lda    #0             standard input
006D 108E0101            ldy    #257           size of buffer (include <CR>)
     *  X points at the start of the buffer already
0071 103F8B              OS9    I$ReadLn       get a command line
     *  X points at the start of the command line
     *  Y contains the length of the command
0074 3420                pshs   Y              save command length on the sta
     *****************
     *  X points at the command
     *  ,S gives the command length
     *
0076              GotParam
0076 3410                pshs   X              save command pointer
     *  stack now contains (ptr, length)
0078 170111              lbsr   SetIcpt
007B 308DFF99            leax   Pipe,PCR
007F 8603                lda    #UPDAT.        open the pipe for update
0081 103F84              OS9    I$Open
0084 10250114            lbcs   Error
0088 9703                sta    PipeNo
008A 8601                lda    #1             std. out path number
008C 103F82              OS9    I$Dup          dup std. out
008F 10250109            lbcs   Error
0093 9704                sta    StdOSave
0095 8601                lda    #1
0097 103F8F              OS9    I$Close        close std. out
009A 102500FE            lbcs   Error
009E 9603                lda    PipeNo         get Pipe path number
00A0 103F82              OS9    I$Dup          Dup it into path 1
     ***********
     *  fork a SHELL with the command line
     *
00A3 3526                puls   D,Y            get command pointer and size
00A5 3440                pshs   U              save U
00A7 1F03                tfr    D,U            command ptr in U
00A9 8600                lda    #0             any language/type
```

```
00AB C601              ldb    #1          one extra page for command lin
00AD 308DFF6C          leax   Shell,PCR
00B1 103F03            OS9    F$Fork      fork the shell
00B4 3540              puls   U           recover U
00B6 102500E2          lbcs   Error
00BA 9705              sta    ChildNo     save Child's process number
     ***********
     *  Get std output back
     *
00BC 8601              lda    #1          std output
00BE 103F8F            OS9    I$Close
00C1 102500D7          lbcs   Error
00C5 9604              lda    StdOSave
00C7 103F82            OS9    I$Dup       dup saved std. out into path 1
00CA 102500CE          lbcs   Error
00CE 9604              lda    StdOSave
00D0 103F8F            OS9    I$Close     close Std. out dup
     *****************
     *  Everything's set up
     *  Now we mostly just copy:
     *
00D3 2023              bra    NxtPage
00D5           Loop
00D5 0D06              tst    Signal
00D7 262A              bne    Quit
00D9 17009F            lbsr   WCRLF       skip to next line
00DC 8D28              bsr    CpyPage     copy a page pipe to std out
00DE 8D4A              bsr    WPrompt     prompt "More:"
00E0 8D59              bsr    GetChar     get reply
00E2 8D69              bsr    CodeChar    change char to 0,2,4,6
00E4 308D0002          leax   JTable,PCR
00E8 6E86              jmp    A,X
00EA           JTable
00EA 2006              bra    NxtLine
00EC 200A              bra    NxtPage
00EE 200F              bra    RunFree
00F0 2011              bra    Quit
00F2           NxtLine
00F2 8601              lda    #1
00F4 9701              sta    CPageLen
00F6 20DD              bra    Loop
00F8           NxtPage
00F8 9600              lda    PPageLen    copy path page length
00FA 4A                deca               minus one
00FB 9701              sta    CPageLen    to copy page length
00FD 20D6              bra    Loop
00FF           RunFree
00FF 0F01              clr    CPageLen
0101 20D2              bra    Loop
0103           Quit
0103 160095            lbra   Exit
     **************
     *  read a page from pipe/
     *  write it to std. out
     *
0106           CpyPage
0106 D601              ldb    CPageLen
0108 32E9FF01          leas   -255,S      temp space
010C 30E4              leax   ,S
010E           CpyLoop
010E 108E00FF          ldy    #255        max length to read
0112 9603              lda    PipeNo
0114 103F8B            OS9    I$ReadLn
0117 10250081          lbcs   Error
011B 8601              lda    #1          std. output
     * X and Y are OK
```

```
011D 103F8C             OS9   I$Writln
0120 257A               bcs   Error
0122 5A                 decb
0123 26E9               bne   CpyLoop
0125 32E900FF           leas  255,S        clear stack
0129 39                 rts                return
     ******************
     *   Write the More prompt
     *
012A           WPrompt
012A 3432               pshs  A,X,Y        save start of command buffer
012C 8601               lda   #1           Standard input
012E 108E0006           ldy   #PromptL
0132 308DFEDC           leax  Prompt,PCR
0136 103F8A             OS9   I$Write
0139 35B2               puls  A,X,Y,PC     return
     ********************
     *   Get a character from std. in
     *   return it in A
     *
013B           GetChar
013B 327F               leas  -1,S         make one byte space on stack
013D 30E4               leax  ,S
013F 8600               lda   #0           std. in
0141 108E0001           ldy   #1           length to read
0145 103F89             OS9   I$Read
0148 2552               bcs   Error
014A 3502               puls  A            get byte from work area
014C 39                 rts                return
     *********************
     *  Code the  character in A
     *  space -> 0  one line
     *  return -> 2 one page
     *  line feed -> 4 run free
     *  tab -> 6 quit
     *  All other characters return 0 (one line)
     *  return the result in A
     *
014D           CodeChar
014D 8120               cmpa  #$20
014F 2302               bls   CodeOK
0151 8620               lda   #$20
0153           CodeOK
0153 308D0003           leax  CodeTbl,PCR
0157 A686               lda   A,X
0159 39                 rts
015A           CodeTbl
015A 00000000           fcb   0,0,0,0,0,0,0,0,0,6,4,0,0,2,0,0
016A 00000000           fcb   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
017A 00                 fcb   0
017B           WCRLF
017B 860D               lda   #$0D         <CR>
017D 3402               pshs  A
017F 108E0001           ldy   #1           length
0183 30E4               leax  ,S
0185 8601               lda   #1
0187 103F8C             OS9   I$WritLn
018A 3582               puls  A,PC         clear stack and return
018C           SetIcpt
018C 308D0004           leax  Intcpt,PCR
0190 103F09             OS9   F$Icpt
0193 39                 rts
0194           Intcpt
0194 C101               cmpb  #S$Wake      is the signal a fatal one?
0196 2302               bls   IntcptX      no; don't change Signal
0198 E746               stb   Signal,U     save signal
```

```
019A              IntcptX
019A 3B                     rti                  return to OS-9
019B              Exit
019B 5F                     clrb
019C              Error
019C 8D03                   bsr    ClnUp
019E 103F06                 OS9    F$Exit
01A1              ClnUp
01A1 3405                   pshs   CC,B
01A3 9605                   lda    ChildNo
01A5 C600                   ldb    #S$KILL
01A7 103F08                 OS9    F$Send        kill our child!
     **********
     *  set Pause back the way it was
     *
01AA 32E8E0                 leas   -32,S
01AD 8601                   lda    #1            std out
01AF C600                   ldb    #SS.Opt       getstat code
01B1 30E4                   leax   ,S
01B3 103F8D                 OS9    I$GetStt
01B6 2509                   bcs    ClnUpX
01B8 9602                   lda    Pause         get saved pause value
01BA A707                   sta    PD.PAU-PD.OPT,X replace it in the option
01BC 8601                   lda    #1            std out
     * B is still SS.Opt from GetStat
     * X still points to the temp buffer
01BE 103F8E                 OS9    I$SetStt      put pause back the way it was
01C1              ClnUpX
01C1 32E820                 leas   32,S
01C4 3585                   puls   CC,B,PC       return
01C6 9E02EC                 EMOD
01C9              MSize     equ    *

00000 error(s)
00000 warning(s)
$01C9 00457 program bytes generated
$0165 00357 data bytes allocated
$23C5 09157 bytes used for symbols
```

nice

A PROGRAM THAT SETS PRIORITIES

Sometimes it is inconveniant to use SETPR to change the priority of a program after it has been started. The following program, Nice, will give you a way to start a program at a priority other than the default. If you give the command:

OS9:nice backup

the backup command will be run at a priority of 0. Nice can also be used to raise a new process's priority. The command

OS9:nice +ds

would start *DynaStar* with a priority of 200. This program uses a different trick for cirumventing the shell's editing of the command line than the one More used. In Nice, each character that has special meaning to the shell has a synonym. The table of syn-

onyms appears in the comments at the beginning of the program. The problem with this trick is that it isn't good to have to remember two different sets of special characters for the shell. The good thing is that this trick makes Nice work like other OS-9 commands.

```
************************ Nice ******************
                             nam   Nice
                             ttl   run a program with a non-standard priorit
                             IFP1
                             ENDC
          *-----------------------------------------------------*
          *      Run a shell command with altered priority.      *
          *      By default Nice will run the program with a     *
          *      priority of 0.                                  *
          *      If the first character in the command is a +    *
          *      the program will be run with a priority of 200. *
          *                                                      *
          *      To permit the shell command line special        *
          *      characters to be passed through to the shell    *
          *      that will execute the command, they are all     *
          *      replaced with alternate character sequences.    *
          *      \:       ;                                      *
          *      \%       &                                      *
          *      \.       !                                      *
          *      \$       #                                      *
          *      \[       (                                      *
          *      \]       )                                      *
          *      \*       >                                      *
          *      \^       <                                      *
          *    In most cases I/O redirection of nice will carry  *
          *    through to the program it runs without using      *
          *    special characters for redirection.               *
          *-----------------------------------------------------*
   0011               Type      set   Prgrm+Objct
   0081               Revs      set   ReEnt+1
   0000 87CD00BE                mod   MSize,Name,Type,Revs,Entry,DSize
 D 0000               Prty      rmb   1
 D 0001               CmdSize   rmb   2
 D 0003               Cmd       rmb   256
 D 0103                         rmb   200         stack space
 D 01CB               DSize     equ   .
   000D 4E6963E5      Name      fcs   /Nice/
   0011 01                      fcb   1           version
   0012 5368656C      Shell     fcs   /Shell/
   00C8               HighP     set   200
   0017               Entry
   0017 0F00                    clr   Prty
   0019 A684                    lda   ,X          check first byte in command li
   001B 812B                    cmpa  #'+         high priority
   001D 2606                    bne   PSet
   001F 3001                    leax  1,X
   0021 86C8                    lda   #HighP      High Priority
   0023 9700                    sta   Prty
   0025               PSet
   0025 8D29                    bsr   BldCmd      copy command to command area
   0027 103F0C                  OS9   F$ID        get process ID
   002A D600                    ldb   Prty
   002C 103F0D                  OS9   F$SPrior    set priority for us
          ***********
          *   now fork a shell
          *
   002F 8600                    lda   #0          any type
   0031 C601                    ldb   #1          one extra page for the command
   0033 308DFFDB                leax  Shell,PCR
   0037 109E01                  ldy   CmdSize
```

```
003A 3440                  pshs   U
003C 3343                  leau   Cmd,U
003E 103F03                OS9    F$Fork
0041 3540                  puls   U
0043 2507                  bcs    Error
0045 103F04                OS9    F$Wait      Wait for the Shell to finish
0048 5D                    tstb               Check return code from Shell
0049 2601                  bne    Error       non-zero ; report error
004B            Exit
004B 5F                    clrb               clear carry
004C            Error
004C 43                    coma               set carry
004D 103F06                OS9    F$Exit      Done
     *************
     *   On entry to BldCmd X points at the command line
     *   On exit the command line has been moved to Cmd,
     *   its size is in CmdSize, and any special characters
     *   in the command have been transformed.
     *
0050            BldCmd
0050 3143                  leay   Cmd,U
0052 0F01                  clr    CmdSize
0054 0F02                  clr    CmdSize+1
0056            BLoop
0056 DC01                  ldd    CmdSize
0058 C30001                addd   #1
005B DD01                  std    CmdSize     keep Size of Cmd updated
005D 8D07                  bsr    LdChar      get a character in A
005F A7A0                  sta    ,Y+         put it in the command buffer
0061 810D                  cmpa   #$0D        <CR>
0063 26F1                  bne    BLoop
0065 39                    rts
0066            LdChar
0066 A680                  lda    ,X+
0068 815C                  cmpa   #'\         Special Escape?
006A 264E                  bne    LdCharX
006C A684                  lda    ,X
006E 813A                  cmpa   #':
0070 2604                  bne    LC1
0072 863B                  lda    #';
0074 2042                  bra    LCRep
0076 8125       LC1        cmpa   #'%
0078 2604                  bne    LC2
007A 8626                  lda    #'&
007C 203A                  bra    LCRep
007E 812E       LC2        cmpa   #'.
0080 2604                  bne    LC3
0082 8621                  lda    #'!
0084 2032                  bra    LCRep
0086 8124       LC3        cmpa   #'$
0088 2604                  bne    LC4
008A 8623                  lda    #'#
008C 202A                  bra    LCRep
008E 815B       LC4        cmpa   #'[
0090 2604                  bne    LC5
0092 8628                  lda    #'(
0094 2022                  bra    LCRep
0096 815D       LC5        cmpa   #']
0098 2604                  bne    LC6
009A 8629                  lda    #')
009C 201A                  bra    LCRep
009E 812A       LC6        cmpa   #'*
00A0 2604                  bne    LC7
00A2 863E                  lda    #'>
00A4 2012                  bra    LCRep
00A6 815E       LC7        cmpa   #'^
```

```
00A8 2604                 bne    LC8
00AA 863C                 lda    #'<
00AC 200A                 bra    LCRep
00AE 815C       LC8       cmpa   #'\
00B0 2602                 bne    LCSkip
00B2 2004                 bra    LCRep
00B4            LCSkip
00B4 865C                 lda    #'\
00B6 2002                 bra    LdCharX
00B8            LCRep
00B8 3001                 leax   1,X          skip over escaped character
00BA            LdCharX
00BA 39                   rts
00BB 8BBEE6               EMOD
00BE            MSize     equ    *

00000 error(s)
00000 warning(s)
$00BE 00190 program bytes generated
$01CB 00459 data bytes allocated
$2311 08977 bytes used for symbols
```

WORKSHOP 4

a null device

Running a program in background without doing something about its input is a mistake you won't make often. Two programs sending lines of output to your screen at the same time can produce baffling results. The normal solution to this problem is to redirect the standard output of the background program to a disk file. This solution works fine and provides a hedge against deciding later that you'd like to see the output. Still, it isn't convenient to have to remember to delete the disk file, and extra disk I/O is always to be avoided.

What is required is a "bit bucket," a place where data can be sent that makes it silently disappear. OS-9 lets us construct a bit bucket without any special hardware. "Bit bucket" is on the long side for a device name and lacks dignity. I use the name "null," abbreviated /nl.

A device that soaks up data as fast as you can send it is useful enough to be worth a little memory space, but it doesn't cost more than a few extra bytes to do something about input, as well. The question is what? Two possibilities come to mind: return some selected character for every read, or, return an EOF error. Since I couldn't decide what character to return, I settled on end-of-file.

The device driver for nl is the simplest SCF device driver I know of.

If you type **nl** and **dmy** into one file, the assembler will leave

both modules in the same file in your execution directory. For those who add the device to their boot file immediately, this isn't important. For others, load the file, then link to the nl (the second module in the file). This method isn't as memory efficient as including the modules in your boot, but it keeps them out of your memory until they are needed.

This program was first published in '68' Micro Journal in April 1983.

******************* Null Device *******************

```
                                   NAM   Dummy I/O driver
                                   TTL   Definitions
            *-----------------------------------------------------*
            *  Dummy                             1July82 Peter Dibble *
            *  return end of file to any read                     *
            *  Put any output down the bit bucket.                *
            *  No error returns                                   *
            *  Public Domain software as of 19Feb83.              *
            *-----------------------------------------------------*
                                   IFP1
                                   use /D0/DEFS/Defslist
                                   ENDC
     00E1                Type      set   DRIVR+OBJCT
     0082                Revs      set   REENT+2
     0000 87CD002E                 MOD   Dummy1,DumNam,Type,Revs,Entry,Memsize
   D 001D                          ORG   V.SCF        leave space for SCFman overhead etc.
   D 001D                Memsize   equ   .
     000D 07                       fcb   READ.+WRITE.+EXEC. driver mode
                                   TTL   Dummy I/O Driver
     000E 446DF9         DumNam    fcs   /Dmy/
     0011 01                       fcb   1            Edition number
     0012                Entry
   W 0012 16000F                   lbra  Init
   W 0015 16000E                   lbra  Read
   W 0018 160009                   lbra  Write
   W 001B 160006                   lbra  GetStat
   W 001E 160003                   lbra  PutStat
   W 0021 160000                   lbra  Term


     0024                Init
     0024                Write
     0024                GetStat
     0024                PutStat
     0024                Term
     0024 5F                       clrb               zero return code
     0025 39                       rts                Do nothing
     0026                Read
     0027 53                       comb               set carry flag
     0028 C6D3                     ldb   #E$EOF       return end of file
     002A 39                       rts                return
     002B 848D35                   emod
     002E                Dummy1    equ   *
                                   TTL   Device Descriptor
            *------------------------------------------------*
            *  NL device descriptor                          *
            *------------------------------------------------*
     00F1                Type      set   DEVIC+OBJCT
     0000 87CD001E                 mod   DDend,DDNam,Type,Revs,FMNam,DRVNam
     000D 07                       fcb   READ.+WRITE.+EXEC. modes
     000E FF0000                   fcb   $FF,0,0      PORT ADDRESS OF 0
```

```
0011 0100                fcb   1,DT.SCF   Options
0013 4ECC       DDNam    fcs   /NL/       device name
0015 5343C6     FMNam    fcs   /SCF/      File Manager Name
0018 446DF9     DRVNam   fcs   /Dmy/
001B BD5979              emod
001E            DDend    equ   *

00000 error(s)
00006 warning(s)
$004C 00076 program bytes generated
$0000 00000 data bytes allocated
$225D 08797 bytes used for symbols
```

a level one acia driver

The following program is the actual acia driver distributed with OS-9 Level One systems. Microware has generously given us permission to publish it, but I must warn you to use this carefully.

There is no guarantee that future versions of OS-9 will support this device driver. This works with OS-9 version 1.2. If it doesn't work with version 1.3 — sorry.

********************** ACIA Driver ******************

```
                              nam   ACIA
                              ttl   Interrupt-Driven Acia driver

           * Copyright 1982 by Microware Systems Corporation
           * Reproduced Under License

           * This source code is the proprietary confidential property of
           * Microware Systems Corporation, and is provided to licensee
           * solely  for documentation and educational purposes. Reproduction,
           * publication, or distribution in any form to any party other than
           * the licensee is strictly prohibited!

                              use   defsfile
0002                LEVEL     equ   2
                              ifpl
                              endc
0001                ByteType set    1
0002                BlockTyp set    2
0002                RamCheck set    BlockTyp
0001                UnLimitd set    1
0002                Limited  set    2
0001                ROMCheck set    UnLimitd
```

```
           ***************
           * Edition History

           *  #   date    Comments
           * -- -------- ----------------------------------------------
           *  3 83/02/11  Add X-ON/X-OFF generation code
           *  4 83/03/10  Getstat Ready returns (B)=bytecount

  0004               Edition  equ   4          Current Edition

           ***************
           * Interrupt-driven Acia Device Driver

  0050               INPSIZ   set   80         input  buffer size (<=256)
  008C               OUTSIZ   set   140        output buffer size (<=256)

  0080               IRQReq   set   %10000000  Interrupt Request
  0040               PARITY   set   %01000000  parity  error bit
  0020               OVERUN   set   %00100000  overrun error bit
  0010               FRAME    set   %00010000  framing error bit
  0008               NOTCTS   set   %00001000  not clear to send
  0004               DCDLST   set   %00000100  data carrier lost

  0080               IRQIN    equ   %10000000  input IRQ enable
  0020               IRQOUT   equ   %00100000  output IRQ enable

  007C               INPERR   set   PARITY+OVERUN+FRAME+NOTCTS+DCDLST

           **********
           * Static storage offsets
           *
D 001D                        org   V.SCF      room for scf variables
D 001D               INXTI    rmb   1          input  buffer next-in  ptr
D 001E               INXTO    rmb   1          input  buffer next-out ptr
D 001F               INCNT    rmb   1          input char count
D 0020               ONXTI    rmb   1          output buffer next-in  ptr
D 0021               ONXTO    rmb   1          output buffer next-out ptr
D 0022               HALTED   rmb   1          output IRQ's disabled when non
D 0023               INHALT   rmb   1          input halted
D 0024               INPBUF   rmb   INPSIZ     input  buffer
D 0074               OUTBUF   rmb   OUTSIZ     output buffer
D 0100               ACIMEM   equ   .          Total static storage requireme

           * HALTED state conditions
  0001               H.XOFF   equ   1          V.XOFF char has been received;
  0002               H.EMPTY  equ   2          Output buffer is empty

           ***************
           * Module Header
  0000 87CD024D               mod   ACIEND,ACINAM,DRIVR+OBJCT,REENT+1,ACIENT,
  000D 03                     fcb   UPDAT.
  000E 414349C1      ACINAM   fcs   "ACIA"

  0012 04                     fcb   Edition    Current Revision

W 0013 160012        ACIENT   lbra  INIT
W 0016 16005D                 lbra  READ
  0019 1600CC                 lbra  WRITE
  001C 1600F9                 lbra  GETSTA
  001F 160109                 lbra  PUTSTA
  0022 16010D                 lbra  TRMNAT

  0025 00            ACMASK   fcb   0          no flip bits
  0026 80                     fcb   IRQReq     Irq polling mask
  0027 0A                     fcb   10         (higher) priority
```

```

                                ttl   'INTERRUPT-DRIVEN Acia device routines
        ***************
        * Init
        *   Initialize (Terminal) Acia
        *
 0028 AE41        INIT     ldx    V.PORT,U    I/o port address
 002A C603                 ldb    #$03        master reset signal
 002C E784                 stb    0,X         reset acia
 002E C602                 ldb    #H.EMPTY
 0030 E7C822               stb    HALTED,U    output IRQ's disabled; buffer
 0033 A6A811               lda    M$OPT,Y     option byte count
 0036 8114        INIT05   cmpa   #PD.PAR-PD.OPT acia control value given?
 0038 2505                 blo    INIT10      ..no; default $15
 003A E6A826               ldb    PD.PAR-PD.OPT+M$DTYP,Y
 003D 2602                 bne    INIT20
 003F C615        INIT10   ldb    #$15        default acia control
 0041 E746        INIT20   stb    V.TYPE,U    save device type
 0043 E784                 stb    0,X         init acia
 0045 A601                 lda    1,X
 0047 A601                 lda    1,X         remove any interrupts
 0049 6D84                 tst    0,X         interrupt gone?
 004B 2B6F                 bmi    ErrNtRdy    ..No; abort
 004D 4F                   clra
 004E 5F                   clrb
 004F EDC81D               std    INXTI,U     Initialize buffer ptrs
 0052 EDC820               std    ONXTI,U
 0055 A7C823               sta    INHALT,U    flag input not halted
 0058 A7C81F               sta    INCNT,U     clear in char count
 005B EC41                 ldd    V.PORT,U
 005D 308DFFC4             leax   ACMASK,PCR
 0061 318D00ED             leay   ACIRQ,PCR   address of interrupt service r
 0065 103F2A               OS9    F$IRQ       Add to IRQ polling table
 0068 2509                 bcs    INIT9       Error - return it
 006A AE41                 ldx    V.PORT,U
 006C E646                 ldb    V.TYPE,U
 006E CA80                 orb    #IRQIN      enable acia input interrupts
 0070 E784                 stb    0,X         initialize acia for input inte
 0072 5F                   clrb
 0073 39          INIT9    rts

        ***************
        * Read
        *   return One Byte of input from the Acia
        *
        * Passed: (Y)=Path Descriptor
        *         (U)=Static Storage address
        * returns: (A)=input Byte (carry clear)
        *     or   CC=Set, B=Error code if error
        *
 0074 8D4A        READ00   bsr    ACSLEP
 0076 A6C823      READ     lda    INHALT,U    is input halted?
 0079 2F15                 ble    Read.a      branch if not
 007B E6C81F               ldb    INCNT,U     get input character count
 007E C10A                 cmpb   #10         less than 10 chars in buffer?
 0080 220E                 bhi    Read.a      branch if not
 0082 E64F                 ldb    V.XON,U     get X-ON char
 0084 CA80                 orb    #Sign       set sign bit
 0086 E7C823               stb    INHALT,U    flag input resume
 0089 E646                 ldb    V.TYPE,U    get control value
 008B CAA0                 orb    #IRQIN!IRQOUT enable input & output IRQs
 008D E7D801               stb    [V.PORT,U]  set control register
 0090 E6C81E      Read.a   ldb    INXTO,U     (input buffer) next-out ptr
 0093 30C824               leax   INPBUF,U    address of input buffer
 0096 1A50                 orcc   #IntMasks   calm interrupts
 0098 E1C81D               cmpb   INXTI,U     any data available?
```

```
009B 27D7                beq   READ00
009D 3A                  abx
009E A684                lda   0,X        the char
00A0 6AC81F              dec   INCNT,U    decrement char count
00A3 5C                  incb             ADVANCE Next-out ptr
00A4 C14F                cmpb  #INPSIZ-1  end of circular buffer?
00A6 2301                bls   READ10     ..no
00A8 5F                  clrb             reset ptr to start of buffer
00A9 E7C81E     READ10   stb   INXTO,U    save updated buffer ptr
00AC 5F                  clrb
00AD E64E                ldb   V.ERR,U    Transmission error?
00AF 2708                beq   READ90     ..no; return
00B1 E7A83A              stb   PD.ERR,Y   return error bits in pd
00B4 6F4E                clr   V.ERR,U
00B6 53                  comb             return carry set
00B7 C6F4                ldb   #E$Read    signal read error
00B9 1CAF       READ90   andcc #^IntMasks enable IRQ requests
00BB 39                  rts

00BC 53         ErrNtRdy comb
00BD C6F6                ldb   #E$NotRdy
00BF 39                  rts

     **********
     * Acslep - Sleep for I/O activity
     *  This version Hogs Cpu if signal pending
     *
     * Passed: (cc)=Irq's Must be disabled
     *         (U)=Global Storage
     *         V.Busy,U=current proc id
     * Destroys: possibly Pc
00C0 3416       ACSLEP   pshs  D,X
00C2 A644                lda   V.BUSY,U   get current process id
00C4 A745                sta   V.Wake,U   arrange wake up signal
00C6 1CAF                andcc #^IntMasks interrupts ok now
00C8 8E0000              ldx   #0
00CB 103F0A              OS9   F$Sleep    wait for input data
00CE 9E50                ldx   D.Proc
00D0 E68819              ldb   P$Signal,X signal present?
00D3 2704                beq   ACSL90     ..no; return
00D5 C103                cmpb  #S$Intrpt  Deadly signal?
00D7 2309                bls   ACSLER     ..yes; return error
00D9 4F         ACSL90   clra             clear carry
00DA A60C                lda   P$State,X  check process state flags
00DC 8502                bita  #Condem    has process died?
00DE 2602                bne   ACSLER     ..Yes; return error
00E0 3596                puls  D,X,PC     return

00E2 3266       ACSLER   leas  6,S        Exit to caller's caller
00E4 43                  coma             return carry set
00E5 39                  rts

     ***************
     * Write
     *   Write char Through Acia
     *
     * Passed: (A)=char to write
     *         (Y)=Path Descriptor
     *         (U)=Static Storage address
     * returns: CC=Set If Busy (output buffer Full)
     *
00E6 8DD8       WRIT00   bsr   ACSLEP     sleep a bit
00E8 30C874     WRITE    leax  OUTBUF,U   output buffer address
00EB E6C820              ldb   ONXTI,U    (output) next-out ptr
00EE 3A                  abx
00EF A784                sta   0,X        Put char in buffer
```

```
  00F1 5C                  incb                ADVANCE the ptr
  00F2 C18B                cmpb   #OUTSIZ-1    end of circular buffer?
  00F4 2301                bls    WRIT10       ..no
  00F6 5F                  clrb                reset ptr to start of buffer
  00F7 1A50       WRIT10   orcc   #IntMasks    disable interrupts
  00F9 E1C821              cmpb   ONXTO,U      buffer full?
  00FC 27E8                beq    WRIT00       ..yes; sleep and retry
  00FE E7C820              stb    ONXTI,U      save updated next-in ptr
  0101 A6C822              lda    HALTED,U     output already enabled?
  0104 270E                beq    Write80      ..yes; don't re-enable
  0106 84FD                anda   #^H.EMPTY    no longer halted due to empty
  0108 A7C822              sta    HALTED,U
  010B 2607                bne    Write80      ..Still HALTED; don't enable I
  010D A646                lda    V.TYPE,U     Parity control
  010F 8AA0                ora    #IRQIN!IRQOUT enable input & output IRQs
  0111 A7D801              sta    [V.PORT,U] Enable output interrupts
  0114 1CAF       Write80  andcc  #^IntMasks enable IRQs
  0116 5F         Write90  clrb                (return carry clear)
  0117 39                  rts

                ***************
                * Getsta/Putsta
                *   Get/Put Acia Status
                *
                * Passed: (A)=Status Code
                *         (Y)=Path Descriptor
                *         (U)=Static Storage address
                * returns: varies
  0118 8101       GETSTA   cmpa   #SS.Ready    Ready status?
  011A 260B                bne    GETS10       ..no
  011C E6C81F              ldb    INCNT,U      get input character count
  011F 279B                beq    ErrNtRdy     ..No; return not ready error
  0121 AE26                ldx    PD.RGS,Y
  0123 E702                stb    R$B,X        return bytecount to caller (!)
  0125 5F         STATUS99 clrb
  0126 39                  rts

  0127 8106       GETS10   cmpa   #SS.EOF      End of file?
  0129 27EB                beq    Write90      ..yes; return carry clear

  012B 53         PUTSTA   comb                return carry set
  012C C6D0                ldb    #E$UnkSvc    Unknown service code
  012E 39                  rts

                ***************
                * Subroutine TRMNAT
                *   Terminate Acia processing
                *
                * Passed: (U)=Static Storage
                * returns: Nothing
                *
W 012F 17FF8E     TRMN00   lbsr   ACSLEP       wait for I/O activity
  0132 9E50       TRMNAT   ldx    D.Proc
  0134 A684                lda    P$ID,X
  0136 A744                sta    V.BUSY,U
  0138 A743                sta    V.LPRC,U
  013A E6C820              ldb    ONXTI,U
  013D 1A50                orcc   #IntMasks    disable interrupts
  013F E1C821              cmpb   ONXTO,U      output done?
  0142 26EB                bne    TRMN00       ..no; sleep a bit
  0144 A646                lda    V.TYPE,U
  0146 A7D801              sta    [V.PORT,U] disable acia interrupts
  0149 1CAF                andcc  #^IntMasks enable interrupts
  014B 8E0000              ldx    #0
  014E 103F2A              OS9    F$IRQ        remove acia from polling tbl
  0151 39                  rts
```

```

       ***************
       * AcIRQ
       *   process Interrupt (input or output) from Acia
       *
       * Passed: (U)=Static Storage addr
       *         (X)=Port address
       *         (A)=polled status
       * Returns: Nothing
       *
0152 AE41      ACIRQ    ldx   V.PORT,U    get port address
0154 1F89               tfr   A,B         copy status
0156 C47C               andb  #INPERR     mask status error bits
0158 EA4E               orb   V.ERR,U
015A E74E               stb   V.ERR,U     update cumulative errors
015C 8505               bita  #5          input ready (or carrier lost)?
015E 264E               bne   InIRQ       ..yes; go get it
       * Fall Through to Do output

       ****************
       * OutIRQ
       *   output to Acia Interrupt Routine
       *
       * Passed: (A)=Acia Status Register Contents
       *         (X)=Acia port address
       *         (U)=Static Storage address

0160 A6C823    OutIRQ   lda   INHALT,U    send X-ON or X-OFF?
0163 2A10               bpl   OutI.a      branch if not
0165 847F               anda  #^Sign      clear sign bit
0167 A701               sta   1,X         send character
0169 A84F               eora  V.XON,U     get zero if X-ON
016B A7C823             sta   INHALT,U    mark it sent
016E A6C822             lda   HALTED,U    is output halted?
0171 2628               bne   OutIRQ3     branch if so
0173 5F                 clrb              clear carry
0174 39                 rts
0175 31C874    OutI.a   leay  OUTBUF,U    output buffer ptr
0178 E6C821             ldb   ONXTO,U     (output) next-out ptr
017B E1C820             cmpb  ONXTI,U     output buffer already empty?
017E 2713               beq   OutIRQ2     ..yes; disable output IRQ, ret
0180 4F                 clra
0181 A6AB               lda   D,Y         next output char
0183 5C                 incb              ADVANCE Next-out ptr
0184 C18B               cmpb  #OUTSIZ-1   end of circular buffer?
0186 2301               bls   OutIRQ1     ..no
0188 5F                 clrb
0189 E7C821    OutIRQ1  stb   ONXTO,U     save updated next-out ptr
018C A701               sta   1,X         Write the char
018E E1C820             cmpb  ONXTI,U     last char in output buffer?
0191 260E               bne   WAKEUP      ..no
0193 A6C822    OutIRQ2  lda   HALTED,U
0196 8A02               ora   #H.EMPTY
0198 A7C822             sta   HALTED,U
019B E646      OutIRQ3  ldb   V.TYPE,U
019D CA80               orb   #IRQIN      disable output IRQs
019F E784               stb   0,X

01A1 C601      WAKEUP   ldb   #S$Wake     Wake up signal
01A3 A645               lda   V.Wake,U    Owner waiting?
01A5 2705      Wake10   beq   Wake90      ..no; return
01A7 6F45               clr   V.Wake,U
01A9 103F08    SendSig  OS9   F$Send      send signal
01AC 5F        Wake90   clrb              return carry clear
01AD 39                 rts

```

```
              ***************
              * Inacia
              *   process Acia input Interrupt
              *
              * Passed: (A)=Acia Status Register data
              *         (X)=Acia port address
              *         (U)=Static Storage address
              *
              * Notice the Absence of Error Checking Here
              *
01AE A601     InIRQ    lda   1,X          Read input char
01B0 2715              beq   InIRQ1       ..NULL, impossible Ctl Chr
01B2 A14B              cmpa  V.INTR,U     keyboard Interrupt?
01B4 275F              beq   InAbort      ..Yes
01B6 A14C              cmpa  V.QUIT,U     keyboard Quit?
01B8 275F              beq   InQuit       ..Yes
01BA A14D              cmpa  V.PCHR,U     keyboard Pause?
01BC 274F              beq   InPause      ..Yes
01BE A14F              cmpa  V.XON,U      X-ON continue?
01C0 2764              beq   InXON        ..Yes
01C2 A1C810            cmpa  V.XOFF,U     X-OFF Immediate Pause request?
01C5 2771              beq   InXOFF       ..Yes
01C7 30C824   InIRQ1   leax  INPBUF,U     input buffer
01CA E6C81D            ldb   INXTI,U      (input) next-in ptr
01CD 3A                abx
01CE A784              sta   0,X          save char in buffer
01D0 5C                incb               update Next-in ptr
01D1 C14F              cmpb  #INPSIZ-1    end of circular buffer?
01D3 2301              bls   InIRQ2       ..no
01D5 5F                clrb
01D6 E1C81E   InIRQ2   cmpb  INXTO,U      input overrun?
01D9 2608              bne   InIRQ30      ..no; good
01DB C620              ldb   #OVERUN      mark overrun error
01DD EA4E              orb   V.ERR,U
01DF E74E              stb   V.ERR,U
01E1 20BE              bra   WAKEUP       throw away character
01E3 E7C81D   InIRQ30  stb   INXTI,U      update next-in ptr
01E6 6CC81F            inc   INCNT,U
01E9 A6C810   InIRQ4   lda   V.XOFF,U     get X-OFF char
01EC 27B3              beq   WAKEUP       branch if not enabled
01EE E6C81F            ldb   INCNT,U      get input count
01F1 C146              cmpb  #INPSIZ-10   is buffer almost full?
01F3 25AC              blo   WAKEUP       bra if not
01F5 E6C823            ldb   INHALT,U     have we sent XOFF?
01F8 26A7              bne   WAKEUP       yes then don't send it again
01FA 847F              anda  #^Sign       insure sign clear
01FC A7C810            sta   V.XOFF,U
01FF 8A80              ora   #Sign        set sign bit
0201 A7C823            sta   INHALT,U     flag input halt
0204 E646              ldb   V.TYPE,U     get control value
0206 CAA0              orb   #IRQIN!IRQOUT enable input & output IRQs
0208 E7D801            stb   [V.PORT,U]
020B 2094              bra   WAKEUP
              ***************
              * Control character routines
020D AE49     InPause  ldx   V.DEV2,U     get echo device static ptr
020F 27B6              beq   InIRQ1       ..None; buffer char, exit
0211 A708              sta   V.PAUS,X     request pause
0213 20B2              bra   InIRQ1       buffer char, exit
0215 C603     InAbort  ldb   #S$Intrpt    keyboard INTERRUPT signal
```

```
  0217 2002              bra    InQuit10
  
  0219 C602    InQuit    ldb    #S$Abort    Abort signal
  021B 3402    InQuit10  pshs   A           save input char
  021D A643              lda    V.LPRC,U    last process id
W 021F 17FF83            lbsr   Wake10      Send error signal
  0222 3502              puls   A           restore input char
  0224 20A1              bra    InIRQ1      buffer char, exit
  
  0226 A6C822  InXON     lda    HALTED,U
  0229 84FE              anda   #^H.XOFF
  022B A7C822            sta    HALTED,U    enable output
  022E 2606              bne    InXON99     ..exit if otherwise disabled
  0230 A646              lda    V.TYPE,U    parity control
  0232 8AA0              ora    #IRQIN!IRQOUT enable input & output IRQs
  0234 A784              sta    0,X
  0236 5F      InXON99   clrb
  0237 39                rts
  
  0238 A6C822  InXOFF    lda    HALTED,U
  023B 2606              bne    InXOFF10    ..already halted, contin
  023D E646              ldb    V.TYPE,U    get acia control code
  023F CA80              orb    #IRQIN      enable only input IRQs
  0241 E784              stb    0,X
  0243 8A01    InXOFF10  ora    #H.XOFF
  0245 A7C822            sta    HALTED,U    restrict output
  0248 5F                clrb
  0249 39                rts
  
  024A 922C7B            emod               Module Crc
  024D         ACIEND    equ    *
  

00000 error(s)
00004 warning(s)
$024D 00589 program bytes generated
$00E3 00227 data bytes allocated
$2AFA 11002 bytes used for symbols
```

WORKSHOP 6

mcia

A LEVEL TWO ACIA DRIVER WITH ENHANCEMENTS

MCIA is a modified version of the version 1.2 Level Two ACIA device driver. The modifications make this driver well-suited to a modem port. It can send and receive breaks, and opens the hardware up enough so that Baud rate and parity can be changed on the fly.

I didn't want to waste memory on two ACIA drivers, so I modified the Microware ACIA driver in a way that remains hidden until activated by the right SetStat call. When it's not triggered, this driver acts enough like the standard ACIA driver that all my software works with it.

```
*    ____________________________________________    *
*                                                    *
*                      MCIA                          *
*                                                    *
*  Support for <break> is provided via two gimics. A *
*  break is sent by SETSTAT with a function code of  *
*  128. This will start sending a <break> immediate- *
*  ly.The break will be sent until a SETSTAT 129 is  *
*  done. SETSTAT 129 will reinitialize the port and  *
*  empty the output buffer.                          *
*                                                    *
*  In order to deal with situations where framing errors *
*  are likely to occur, other than when a break is sent, *
```

```
*  MCIA defaults to a mode where all framing errors    *
*  are treated as errors. To cause MCIA to start treat- *
*  ing framing errors as breaks, use SETSTAT 130.       *
*  To set the mode back to no-break, use SETSTAT        *
*  131. There is also a GETSTAT (code 129), which       *
*  will return with carry set if the terminal is in break *
*  detect mode.                                         *
*                                                       *
*  A <break> looks almost exactly like an <interrupt>   *
*  when it is received. It sends a code 3 to the con    *
*  trolling task. If a GETSTAT code 128 is done         *
*  before any other codes are sent, it will return with *
*  carry clear to indicate that the last code 3 was     *
*  for a <break>.                                       *
*                                                       *
*  ___________________________________________________  *
*                                                       *
*  GetStat 131 returns the amount of input waiting in   *
*  Y if there is any. It returns with carry set if there *
*  is no input waiting.                                 *
*                                                       *
*  An additional Setstat (132) reinitializes the port.  *
*  This, particularly, is intended to allow V.TYPE to   *
*  be changed if the parity byte is changed in the      *
*  path descriptor.                                     *
*                                                       *
*  These calls are set up to provide the maximum        *
*  chance of having programs, written to work with      *
*  this driver, work with the older drivers.            *
*  ___________________________________________________  *

                 nam   MCIA
                 ttl   Interrupt-Driven Acia driver

* Copyright 1982 by Microware Systems Corporation
* Reproduced Under License

* This source code is the proprietary confidential property of
* Microware Systems Corporation, and is provided to licensee
* solely  for documentation and educational purposes. Reproduction,
* publication, or distribution in any form to any party other than
* the licensee is strictly prohibited!

                 IFP1
                 ENDC

***************
* Edition History

*  #   date     Comments
* -- -------- ----------------------------------------------
*  3 83/02/11 Add X-ON/X-OFF generation code
*  4 83/03/10 Add SS.SSIG putstat code
*  4 83/03/10 Getstat Ready returns (B)=bytecount
*  4 83/03/17 Putstat Release removes SS.SSIG if necessary
*  4 83/03/25 INCNT was counting chars even if buffer overrun.
*  5 83/06/01 Modified to use Suspend process state instead
*             of F$Send during IRQ.
*  6 84/05/18 Support for Break reset of V.TYPE and read
```

```
status.

 0006             Edition  equ   6            Current Edition

       ***************
       * Interrupt-driven Acia Device Driver

 0050             INPSIZ   set   80           input  buffer size (<=256)
 008C             OUTSIZ   set   140          output buffer size (<=256)

 0080             IRQReq   set   %10000000    Interrupt Request
 0040             PARITY   set   %01000000    parity  error bit
 0020             OVERUN   set   %00100000    overrun error bit
 0010             FRAME    set   %00010000    framing error bit
 0008             NOTCTS   set   %00001000    not clear to send
 0004             DCDLST   set   %00000100    data carrier lost

 0060             SBREAK   set   %01100000    send a break

 0080             IRQIN    equ   %10000000    input IRQ enable
 0020             IRQOUT   equ   %00100000    output IRQ enable

       ****************
       *  Added GetStat/PutStat codes for break processing
       *  Added Setstat for reinitializing V.TYPE
       *  Added GetStat to return device status
       *
 0080             SetBrk   set   128          Start sending a break - setsta
 0081             ClrBrk   set   129          Stop sending a break - setstat
 0082             BrkMode  set   130          Set detect-break mode on - set
 0083             ErrMode  set   131          Set detect-break mode off - se
 0084             ReInit   set   132          Reinitialize V.TYPE - setstat

 0080             TstBrk   set   128          Is a break being sent? - getst
 0081             TstBrkM  set   129          In detect-break mode ? - getst
 0082             GetSReg  set   130          Return device status - getstat



 007C             INPERR   set   PARITY+OVERUN+FRAME+NOTCTS+DCDLST

       **********
       * Static storage offsets
       *
D 001D                     org   V.SCF        room for scf variables
D 001D            INXTI    rmb   1            input  buffer next-in  ptr
D 001E            INXTO    rmb   1            input  buffer next-out ptr
D 001F            INCNT    rmb   1            input char count
D 0020            ONXTI    rmb   1            output buffer next-in  ptr
D 0021            ONXTO    rmb   1            output buffer next-out ptr
D 0022            HALTED   rmb   1            output IRQ's disabled when non
D 0023            INHALT   rmb   1            input halted
D 0024            SIGPRC   rmb   2            Process to signal and code
D 0026            LoclFl   rmb   1            local flags
  0080            LFBrkS   equ   $80          break being sent now
  0040            LFBrkR   equ   $40          break has been recieved
  0020            LFBrkN   equ   $20          detect-break mode on
D 0027            INPBUF   rmb   INPSIZ       input  buffer
D 0077            OUTBUF   rmb   OUTSIZ       output buffer
D 0103            ACIMEM   equ   .            Total static storage requireme

       * HALTED state conditions
  0001            H.XOFF   equ   1            V.XOFF char has been received;
  0002            H.EMPTY  equ   2            Output buffer is empty

  0008            Revision set   8
```

```
          ***************
          * Module Header
  0000 87CD0391            mod   ACIEND,ACINAM,DRIVR+OBJCT,REENT+Revision,
  000D 03                  fcb   UPDAT.
  000E 4D4349C1   ACINAM   fcs   "MCIA"

  0012 06                  fcb   Edition     Current Revision

W 0013 160013     ACIENT   lbra  INIT
W 0016 16006D              lbra  READ
  0019 1600F5              lbra  Write
  001C 160132              lbra  GETSTA
  001F 16016D              lbra  PUTSTA
  0022 160203              lbra  TRMNAT

  0025 00         FlagInit fcb   0           initial value for local flags
  0026 00         ACMASK   fcb   0           no flip bits
  0027 80                  fcb   IRQReq      Irq polling mask
  0028 0A                  fcb   10          (higher) priority

                           ttl   INTERRUPT-DRIVEN Acia device routines
          ***************
          * Init
          *   Initialize (Terminal) Acia
          *
  0029 AE41       INIT     ldx   V.PORT,U    I/o port address
  002B C603                ldb   #$03        master reset signal
  002D E784                stb   0,X         reset acia
  002F C602                ldb   #H.EMPTY
  0031 E7C822              stb   HALTED,U    output IRQ's disabled; buffer
          *******
          *  changes by PCD
          *
  0034 A68DFFED            lda   FlagInit,PCR get initial value for LoclFl
  0038 A7C826              sta   LoclFl,U    initialize local flags
  003B 8D36                bsr   SetTYPE     return with initialization byt
          * end of changes, PCD

  003D E784                stb   0,X         init acia
  003F A601                lda   1,X
  0041 A601                lda   1,X         remove any interrupts
  0043 6D84                tst   0,X         interrupt gone?
  0045 102B0094            lbmi  ErrNtRDy    ..No; abort
  0049 4F                  clra
  004A 5F                  clrb
  004B EDC81D              std   INXTI,U     Initialize buffer ptrs
  004E EDC820              std   ONXTI,U
  0051 A7C823              sta   INHALT,U    flag input not halted
  0054 A7C81F              sta   INCNT,U     clear in char count
  0057 EDC824              std   SIGPRC,U    no process to signal
  005A EC41                ldd   V.PORT,U
  005C 308DFFC6            leax  ACMASK,PCR
  0060 318D01E4            leay  ACIRQ,PCR   address of interrupt service r
  0064 103F2A              OS9   F$IRQ       Add to IRQ polling table
  0067 2509                bcs   INIT9       Error - return it
  0069 AE41                ldx   V.PORT,U
  006B E646                ldb   V.TYPE,U
  006D CA80                orb   #IRQIN      enable acia input interrupts
  006F E784                stb   0,X         initialize acia for input inte
  0071 5F                  clrb
  0072 39         INIT9    rts

  0073            SetTYPE
  0073 A6A811              lda   M$OPT,Y     option byte count
  0076 8114                cmpa  #PD.PAR-PD.OPT acia control value given?
  0078 2505                blo   INIT10      ..no; default $15
```

```
007A E6A826                 ldb    PD.PAR-PD.OPT+M$DTYP,Y
007D 2602                   bne    INIT20
007F C615      INIT10       ldb    #$15         default acia control
0081 E746      INIT20       stb    V.TYPE,U     save device type
0083 39                     rts                 return

          ***************
          * Read
          *   return One Byte of input from the Acia
          *
          * Passed: (Y)=Path Descriptor
          *         (U)=Static Storage address
          * returns: (A)=input Byte (carry clear)
          *      or   CC=Set, B=Error code if error
          *
0084 8D5B      READ00       bsr    ACSLEP
0086 A6C823    READ         lda    INHALT,U     is input halted?
0089 2F15                   ble    Read.a       branch if not
008B E6C81F                 ldb    INCNT,U      get input character count
008E C10A                   cmpb   #10          less than 10 chars in buffer?
0090 220E                   bhi    Read.a       branch if not
0092 E64F                   ldb    V.XON,U      get X-ON char
0094 CA80                   orb    #Sign        set sign bit
0096 E7C823                 stb    INHALT,U     flag input resume
0099 E646                   ldb    V.TYPE,U     get control value
009B CAA0                   orb    #IRQIN!IRQOUT enable input & output IRQs
009D E7D801                 stb    [V.PORT,U] set control register
00A0 6DC824    Read.a       tst    SIGPRC,U     a process waiting for device?
00A3 2638                   bne    ErrNtRdy     ..Yes; return dormant terminal
00A5 E6C81E                 ldb    INXTO,U      (input buffer) next-out ptr
00A8 30C827                 leax   INPBUF,U     address of input buffer
00AB 1A50                   orcc   #IntMasks    calm interrupts
00AD E1C81D                 cmpb   INXTI,U      any data available?
00B0 27D2                   beq    READ00
00B2 3A                     abx
00B3 A684                   lda    0,X          the char
00B5 6AC81F                 dec    INCNT,U      decrement char count
00B8 5C                     incb                ADVANCE Next-out ptr
00B9 C14F                   cmpb   #INPSIZ-1    end of circular buffer?
00BB 2301                   bls    READ10       ..no
00BD 5F                     clrb                reset ptr to start of buffer
00BE E7C81E    READ10       stb    INXTO,U      save updated buffer ptr
          ************
          *  Modification to deal with break -- PCD
          *
00C1 E6C826                 ldb    LoclFl,U
00C4 C540                   bitb   #LFbrkR      break received recently
00C6 2705                   beq    Read20       .. no; check for errors
00C8 5F                     clrb
00C9 E74E                   stb    V.Err,U      clear any error
00CB 200D                   bra    READ90
00CD           Read20
          *  end of mod -- PCD

00CD 5F                     clrb
00CE E64E                   ldb    V.ERR,U      Transmission error?
00D0 2708                   beq    READ90       ..no; return
00D2 E7A83A                 stb    PD.ERR,Y     return error bits in pd
00D5 6F4E                   clr    V.ERR,U
00D7 53                     comb                return carry set
00D8 C6F4                   ldb    #E$Read      signal read error
00DA 1CAF      READ90       andcc  #^IntMasks enable IRQ requests
00DC 39                     rts

00DD 53        ErrNtRdy     comb
00DE C6F6                   ldb    #E$NotRdy
```

```
00E0 39                    rts

     **********
     * Acslep - Sleep for I/O activity
     *  This version Hogs Cpu if signal pending
     *
     * Passed: (cc)=Irq's Must be disabled
     *         (U)=Global Storage
     *         V.Busy,U=current proc id
     * Destroys: possibly Pc
00E1 3416         ACSLEP   pshs  D,X
00E3 9650                  lda   D.Proc      get current process ptr
00E5 A745                  sta   V.Wake,U    arrange wake up ptr
00E7 9E50                  ldx   D.Proc      get process ptr
00E9 A60C                  lda   P$State,x   get process state
00EB 8A08                  ora   #Suspend    set suspend bit
00ED A70C                  sta   P$State,x   put it in descriptor
00EF 1CAF                  andcc #^IntMasks interrupts ok now
00F1 8E0001                ldx   #1
00F4 103F0A                OS9   F$Sleep     wait for input data
00F7 9E50                  ldx   D.Proc
00F9 E68819                ldb   P$Signal,X signal present?
00FC 2704                  beq   ACSL90      ..no; return
00FE C103                  cmpb  #S$Intrpt   Deadly signal?
0100 2309                  bls   ACSLER      ..yes; return error
0102 4F           ACSL90   clra              clear carry
0103 A60C                  lda   P$State,X   check process state flags
0105 8502                  bita  #Condem     has process died?
0107 2602                  bne   ACSLER      ..Yes; return error
0109 3596                  puls  D,X,PC      return

010B 3266         ACSLER   leas  6,S         Exit to caller's caller
010D 43                    coma              return carry set
010E 39                    rts

     ***************
     * Write
     *    Write char Through Acia
     *
     * Passed: (A)=char to write
     *         (Y)=Path Descriptor
     *         (U)=Static Storage address
     * returns: CC=Set If Busy (output buffer Full)
     *
010F 8DD0         WRIT00   bsr   ACSLEP      sleep a bit
0111              Write
     *********
     *  Modifications for break support -- PCD
     *
0111 E6C826                ldb   Loc1F1,U
0114 C580                  bitb  #LFbrkS     is a break being sent?
0116 2630                  bne   WRITE95     ..yes; recover from error
     *  end of modification -- PCD

0118 30C877                leax  OUTBUF,U    output buffer address
011B E6C820                ldb   ONXTI,U     (output) next-out ptr
011E 3A                    abx
011F A784                  sta   0,X         Put char in buffer
0121 5C                    incb              ADVANCE the ptr
0122 C18B                  cmpb  #OUTSIZ-1   end of circular buffer?
0124 2301                  bls   WRIT10      ..no
0126 5F                    clrb              reset ptr to start of buffer
0127 1A50         WRIT10   orcc  #IntMasks   disable interrupts
0129 E1C821                cmpb  ONXTO,U     buffer full?
012C 27E1                  beq   WRIT00      ..yes; sleep and retry
012E E7C820                stb   ONXTI,U     save updated next-in ptr
```

```
0131 A6C822              lda    HALTED,U    output already enabled?
0134 270E                beq    Write80     ..yes; don't re-enable
0136 84FD                anda   #^H.EMPTY   no longer halted due to empty
0138 A7C822              sta    HALTED,U
013B 2607                bne    Write80     ..Still HALTED; don't enable I
013D A646                lda    V.TYPE,U    Parity control
013F 8AA0                ora    #IRQIN!IRQOUT enable input & output IRQs
0141 A7D801              sta    [V.PORT,U]  Enable output interrupts
0144 1CAF      Write80   andcc  #^IntMasks  enable IRQs
0146 5F        Write90   clrb               (return carry clear)
0147 39                  rts

0148           WRITE95
0148 3402                pshs   A
014A 170090              lbsr   Puts25
014D 3502                puls   A
014F 20C0                bra    Write

     ***************
     * Getsta/Putsta
     *   Get/Put Acia Status
     *
     * Passed: (A)=Status Code
     *         (Y)=Path Descriptor
     *         (U)=Static Storage address
     * returns: varies
0151 8101      GETSTA    cmpa   #SS.Ready   Ready status?
0153 260B                bne    GETS10      ..no
0155 E6C81F              ldb    INCNT,U     get input character count
0158 2783                beq    ErrNtRdy    ..No; return not ready error
015A AE26                ldx    PD.RGS,Y
015C E702                stb    R$B,X       return bytecount to caller (!)
015E 5F        STATUS99  clrb
015F 39                  rts

0160 8106      GETS10    cmpa   #SS.EOF     End of file?
0162 27E2                beq    Write90     ..yes; return carry clear

     ***********
     *  Mods for break support -- PCD
     *
0164 8180                cmpa   #TstBrk     test for received break?
0166 2609                bne    GetS20
0168 A6C826              lda    LoclFl,U
016B 8540                bita   #LFbrkR     break received?
016D 26D7                bne    Write90
016F           GetS15
016F 43                  coma               set carry
0170 39                  rts                return

0171           GetS20
0171 8181                cmpa   #TstBrkM    test for break mode?
0173 2609                bne    GetS30      .. not equal; check for Read S
0175 A6C826              lda    LoclFl,U
0178 8520                bita   #LFBrkN     break mode on?
017A 26F3                bne    GetS15      yes; return with carry set
017C 20C8                bra    Write90     no; return with carry clear

017E           GetS30
017E 8182                cmpa   #GetSReg    read device status register?
0180 2609                bne    Unknown
0182 E6D801              ldb    [V.Port,U]  read device status register
0185 AE26                ldx    PD.RGS,Y
0187 E702                stb    R$B,X       return status to caller
0189 20BB                bra    Write90
     **********************
```

```
     * End of modifications -- PCD
     *

018B 53        Unknown  comb               return carry set
018C C6D0               ldb    #E$UnkSvc   Unknown service code
018E 39                 rts

018F 811A      PUTSTA   cmpa   #SS.SSIG    signal process when ready?
0191 2617               bne    PUTS.A
0193 A625               lda    PD.CPR,Y    get process id
0195 AE26               ldx    PD.RGS,Y
0197 E605               ldb    R$X+1,X     get signal code
0199 1A50               orcc   #IntMasks   disable IRQs
019B 6DC81F             tst    INCNT,U     any data available?
019E 2605               bne    PUTS10      yes, data ready
01A0 EDC824             std    SIGPRC,U    save the signal data
01A3 209F               bra    Write80     exit

01A5 1CAF      PUTS10   andcc  #^IntMasks  enable IRQs
01A7 160174             lbra   SendSig     send the signal

01AA 811B      PUTS.A   cmpa   #SS.Relea   Release Device?
01AC 260B               bne    PUTS12      bra if not
01AE A625               lda    PD.CPR,Y    current process ID
01B0 A1C824             cmpa   SIGPRC,U    waiting for data?
01B3 26A9               bne    STATUS99    ..no; return carry clear
01B5 6FC824             clr    SIGPRC,U
01B8 39                 rts

01B9           PUTS12
01B9 8180               cmpa   #SetBrk
01BB 261C               bne    PutS20
01BD A6C826             lda    LoclFl,U
01C0 8A80               ora    #LFbrkS        indicate that we're sending a break
01C2 A7C826             sta    LoclFl,U
01C5 1A50               orcc   #IntMasks   disable interrupts
01C7 6FC820             clr    ONXTI,U     empty buffers
01CA 6FC821             clr    ONXTO,U
     *******
     * modify the control registers
     *
01CD A646               lda    V.TYPE,U    it should show input interrupt
01CF 8AE0               ora    #SBREAK!IRQIN and break
01D1 A7D801             sta    [V.PORT,U]
01D4 1CAF               andcc  #^IntMasks
01D6 16FF6D             lbra   Write90     return with B clear
01D9           PutS20
01D9 8181               cmpa   #ClrBrk
01DB 2616               bne    PutS30
01DD           PutS25
01DD 1A50               orcc   #IntMasks
01DF A6C826             lda    LoclFl,U
01E2 847F               anda   #^LFbrkS    indicate that we aren't sending a break
01E4 A7C826             sta    LoclFl,U
01E7 A646               lda    V.TYPE,U
01E9 8A80               ora    #IRQIN
01EB A7D801             sta    [V.PORT,U] shut off the break
01EE 1CAF               andcc  #^IntMasks
01F0 16FF53             lbra   Write90     return with B clear
01F3           PutS30
01F3 8182               cmpa   #BrkMode    set to break detect mode
01F5 260F               bne    PutS40
01F7 1A50               orcc   #IntMasks
01F9 A6C826             lda    LoclFl,U    get local flags
01FC 8A20               ora    #LFBrkN     set break detect on
01FE A7C826             sta    LoclFl,U
```

```
0201 1CAF                   andcc #^IntMasks
0203 16FF40                 lbra  Write90

0206              Puts40
0206 8183                   cmpa  #ErrMode    get out of detect-break mode?
0208 260F                   bne   PutS50
020A 1A50                   orcc  #IntMasks
020C A6C826                 lda   LoclFl,U    get local flags
020F 84DF                   anda  #^LFBrkN    set detect-break mode off
0211 A7C826                 sta   LoclFl,U
0214 1CAF                   andcc #^IntMasks
0216 16FF2D                 lbra  Write90

0219              PutS50
0219 9184                   cmpa  ReInit
021B 1026FF6C               lbne  Unknown
021F 17FE51                 lbsr  SetTYPE
0222 16FF21                 lbra  Write90

     ***************
     * Subroutine TRMNAT
     *   Terminate Acia processing
     *
     * Passed: (U)=Static Storage
     * returns: Nothing
     *
0225 17FEB9       TRMN00    lbsr  ACSLEP      wait for I/O activity
0228 9E50         TRMNAT    ldx   D.Proc
022A A684                   lda   P$ID,X
022C A744                   sta   V.BUSY,U
022E A743                   sta   V.LPRC,U
0230 E6C820                 ldb   ONXTI,U
0233 1A50                   orcc  #IntMasks   disable interrupts
0235 E1C821                 cmpb  ONXTO,U     output done?
0238 26EB                   bne   TRMN00      ..no; sleep a bit
023A A646                   lda   V.TYPE,U
023C A7D801                 sta   [V.PORT,U] disable acia interrupts
023F 1CAF                   andcc #^IntMasks enable interrupts
0241 8E0000                 ldx   #0
0244 103F2A                 OS9   F$IRQ       remove acia from polling tbl
0247 39                     rts

     ***************
     * AcIRQ
     *   process Interrupt (input or output) from Acia
     *
     * Passed: (U)=Static Storage addr
     *         (X)=Port address
     *         (A)=polled status
     * Returns: Nothing
     *
0248 AE41         ACIRQ     ldx   V.PORT,U    get port address
024A 1F89                   tfr   A,B         copy status
024C C47C                   andb  #INPERR     mask status error bits
024E EA4E                   orb   V.ERR,U
0250 E74E                   stb   V.ERR,U     update cumulative errors
0252 8505                   bita  #5          input ready (or carrier lost)?
0254 2652                   bne   InIRQ       ..yes; go get it
     * Fall Through to Do output

     ****************
     * OutIRQ
     *   output to Acia Interrupt Routine
     *
     * Passed: (A)=Acia Status Register Contents
     *         (X)=Acia port address
```

```
           *         (U)=Static Storage address

0256 A6C823    OutIRQ    lda    INHALT,U     send X-ON or X-OFF?
0259 2A10                bpl    OutI.a       branch if not
025B 847F                anda   #^Sign       clear sign bit
025D A701                sta    1,X          send character
025F A84F                eora   V.XON,U      get zero if X-ON
0261 A7C823              sta    INHALT,U     mark it sent
0264 A6C822              lda    HALTED,U     is output halted?
0267 2628                bne    OutIRQ3      branch if so
0269 5F                  clrb                clear carry
026A 39                  rts
026B 31C877    OutI.a    leay   OUTBUF,U     output buffer ptr
026E E6C821              ldb    ONXTO,U      (output) next-out ptr
0271 E1C820              cmpb   ONXTI,U      output buffer already empty?
0274 2713                beq    OutIRQ2      ..yes; disable output IRQ, ret
0276 4F                  clra
0277 A6AB                lda    D,Y          next output char
0279 5C                  incb                ADVANCE Next-out ptr
027A C18B                cmpb   #OUTSIZ-1    end of circular buffer?
027C 2301                bls    OutIRQ1      ..no
027E 5F                  clrb
027F E7C821    OutIRQ1   stb    ONXTO,U      save updated next-out ptr
0282 A701                sta    1,X          Write the char
0284 E1C820              cmpb   ONXTI,U      last char in output buffer?
0287 260E                bne    WakeUp       ..no
0289 A6C822    OutIRQ2   lda    HALTED,U
028C 8A02                ora    #H.EMPTY
028E A7C822              sta    HALTED,U
0291 E646      OutIRQ3   ldb    V.TYPE,U
0293 CA80                orb    #IRQIN       disable output IRQs
0295 E784                stb    0,X

0297 A645      WakeUp    lda    V.Wake,U     Owner waiting?
0299 270B                beq    Wake90       ..no; return
029B 5F                  clrb
029C E745                stb    V.Wake,U
029E 1F01                tfr    d,x          get ptr to process desc
02A0 A60C                lda    P$State,x
02A2 84F7                anda   #^Suspend    clear suspend state
02A4 A70C                sta    P$State,x    put it in proc desc
02A6 5F        Wake90    clrb                return carry clear
02A7 39                  rts

           ***************
           * Inacia
           *   process Acia input Interrupt
           *
           * Passed: (A)=Acia Status Register data
           *         (X)=Acia port address
           *         (U)=Static Storage address
           *
           * Notice the Absence of Error Checking Here
           *
02A8                InIRQ
           *********
           *  Modifications for break support -- PCD
           *  If the framing error bit is on in the status register
           *  we very likely have a <break>
02A8 8510                bita   #FRAME       <break>
02AA 2716                beq    InIRQ5       no; continue
           *****
           *  There is a break
           *
02AC A601                lda    1,X          clear framing error bit
02AE A6C826              lda    LoclFl,U     get local flags
```

```
02B1 C520               bitb   #LFBrkN     detect-break on?
02B3 2721               beq    InIRQ7      no; pretend we never saw it
02B5 C540               bitb   #LFBrkR     have we already seen this brea
02B7 2619               bne    DummyX      ..Yes; just leave
02B9 CA40               orb    #LFbrkR     set break received
02BB E7C826             stb    LoclFl,U
02BE A64B               lda    V.INTR,U    fake a keyboard interrupt
02C0 2014               bra    InIRQ7

02C2           InIRQ5
     ********
     *  If there isn't a break, turn of the
     *  "now receiving a break" bit in LoclFl
     *
02C2 A6C826             lda    LoclFl,U
02C5 8540               bita   #LFBrkR     was a break received?
02C7 270B               beq    InIRQ6
     ******
     *  The character right after a break is generally junk.
     *  Throw it out and clear any error indication
     *
02C9 84BF               anda   #^LFbrkR    clear <break> received flag
02CB A7C826             sta    LoclFl,U
02CE 6F4E               clr    V.ERR,U     clear error byte
02D0 A601               lda    1,X         empty input register
02D2           DummyX
02D2 5F                 clrb               clear carry
02D3 39                 rts
02D4           InIRQ6
02D4 A601               lda    1,X         Read input char
02D6           InIRQ7
02D6 2719               beq    InIRQ1      ..NULL, impossible Ctl Chr
02D8 A14B               cmpa   V.INTR,U    keyboard Interrupt?
02DA 277A               beq    InAbort     ..Yes
02DC A14C               cmpa   V.QUIT,U    keyboard Quit?
02DE 277A               beq    InQuit      ..Yes
02E0 A14D               cmpa   V.PCHR,U    keyboard Pause?
02E2 276A               beq    InPause     ..Yes
02E4 A14F               cmpa   V.XON,U     X-ON continue?
02E6 10270080           lbeq   InXON       ..Yes
02EA A1C810             cmpa   V.XOFF,U    X-OFF Immediate Pause request?
02ED 1027008B           lbeq   InXOFF      ..Yes

02F1 30C827    InIRQ1   leax   INPBUF,U    input buffer
02F4 E6C81D             ldb    INXTI,U     (input) next-in ptr
02F7 3A                 abx
02F8 A784               sta    0,X         save char in buffer
02FA 5C                 incb               update Next-in ptr
02FB C14F               cmpb   #INPSIZ-1   end of circular buffer?
02FD 2301               bls    InIRQ2      ..no
02FF 5F                 clrb
0300 E1C81E    InIRQ2   cmpb   INXTO,U     input overrun?
0303 2608               bne    InIRQ30     ..no; good
0305 C620               ldb    #OVERUN     mark overrun error
0307 EA4E               orb    V.ERR,U
0309 E74E               stb    V.ERR,U
030B 208A               bra    WakeUp      throw away character

030D E7C81D    InIRQ30  stb    INXTI,U     update next-in ptr
0310 6CC81F             inc    INCNT,U
0313 6DC824             tst    SIGPRC,U    any process to notify?
0316 270B               beq    InIRQ4      ..no
0318 ECC824             ldd    SIGPRC,U    get process to signal
031B 6FC824             clr    SIGPRC,U    disable signal sending
031E 103F08    SendSig  OS9    F$Send      send signal
0321 5F                 clrb               return carry clear
```

```
0322 39                  rts

0323 A6C810     InIRQ4   lda   V.XOFF,U     get X-OFF char
0326 1027FF6D            lbeq  WakeUp       branch if not enabled
032A E6C81F              ldb   INCNT,U      get input count
032D C146                cmpb  #INPSIZ-10 is buffer almost full?
032F 1025FF64            lblo  WAKEUP       bra if not
0333 E6C823              ldb   INHALT,U     have we sent XOFF?
0336 1026FF5D            lbne  WAKEUP       yes then don't send it again
033A 847F                anda  #^Sign       insure sign clear
033C A7C810              sta   V.XOFF,U
033F 8A80                ora   #Sign        set sign bit
0341 A7C823              sta   INHALT,U     flag input halt
0344 E646                ldb   V.TYPE,U     get control value
0346 CAA0                orb   #IRQIN!IRQOUT enable input & output IRQs
0348 E7D801              stb   [V.PORT,U]
034B 16FF49              lbra  WakeUp

     ***************
     * Control character routines

034E AE49       InPause  ldx   V.DEV2,U     get echo device static ptr
0350 279F                beq   InIRQ1       ..None; buffer char, exit
0352 A708                sta   V.PAUS,X     request pause
0354 209B                bra   InIRQ1       buffer char, exit

0356 C603       InAbort  ldb   #S$Intrpt    keyboard INTERRUPT signal
0358 2002                bra   InQuit10

035A C602       InQuit   ldb   #S$Abort     Abort signal
035C 3402       InQuit10 pshs  A            save input char
035E A643                lda   V.LPRC,U     last process id
0360 2704                beq   InQuit30     bra if no place to send
0362 6F45                clr   V.Wake,u
0364 8DB8                bsr   SendSig
0366 3502       InQuit30 puls  A            restore input char
0368 2087                bra   InIRQ1       buffer char, exit

036A A6C822     InXON    lda   HALTED,U
036D 84FE                anda  #^H.XOFF
036F A7C822              sta   HALTED,U     enable output
0372 2606                bne   InXON99      ..exit if otherwise disabled
0374 A646                lda   V.TYPE,U     parity control
0376 8AA0                ora   #IRQIN!IRQOUT enable input & output IRQs
0378 A784                sta   0,X
037A 5F         InXON99  clrb
037B 39                  rts

037C A6C822     InXOFF   lda   HALTED,U
037F 2606                bne   InXOFF10     ..already halted, continue
0381 E646                ldb   V.TYPE,U     get acia control code
0383 CA80                orb   #IRQIN       enable only input IRQs
0385 E784                stb   0,X
0387 8A01       InXOFF10 ora   #H.XOFF
0389 A7C822              sta   HALTED,U     restrict output
038C 5F                  clrb
038D 39                  rts

038E 063552              emod               Module Crc
0391            ACIEND   equ   *


00000 error(s)
00002 warning(s)
$0391 00913 program bytes generated
$00E6 00230 data bytes allocated
$27DF 10207 bytes used for symbols
```

WORKSHOP 7

an rbf device driver

Microware has given us permission to print this device driver. It is the most general device driver we could find that wasn't the property of some hardware vendor. This device driver handles a 10 megabyte hard disk using interrupts and DMA. It has all the features you are likely to need in an RBF driver.

I don't have a DTC disk controller on my system, so I didn't try this driver. I present it largely without comment.

One comment I can't resist making: this appears to be an early effort at Microware. Notice that the address of the controller is locked into the code. It would be more in accord with their standards to keep the controller address in the device descriptor, and use offsets from that address in the driver.

```
                   nam   DTC 1403
                   ttl   Driver for DTC 1403 Disk Controller

 ************************************************************
 *
 *
 *    OS-9 Device Driver Module For DTC 1403
 *    Winchester/Floppy Disk Storage Module
 *    Using DTC-68 Host Adaptor
 *
 *    (C) 1982 Microware Systems Corporation
 *    Reproduction or Duplication Prohibited
 *
```

```
        *   Revision: A
        *   Date: 19 April 1982
        *
        ***************************************************************

        * include equate files OS9Defs and RBFDefs
                         ifpl
                         endc

E300            ContPort EQU   $E300      Controller Address

        *
        * Disk Driver Equates
        *

        * Device Port Assignments

E300            CSR.REG  equ   ContPort   device memory address
E301            CIR.REG  equ   CSR.REG+1  Completion Information Registe
E302            DATA.DMA equ   CIR.REG+1  Data DMA Address Register
E304            CMD.DMA  equ   DATA.DMA+2 Command DMA Address Register

        * Controller Command Opcodes

0000            C$TREADY EQU   0          test if ready/no-op
0001            C$RESTOR EQU   1          restore to track 0
0002            C$REQSYN EQU   2          request error syndrome
0003            C$REQDET EQU   3          request error detail
0004            C$FORALL EQU   4          format all sectors
0005            C$FORCHK EQU   5          check track format
0006            C$FORTRK EQU   6          format single track
0007            C$FORBAD EQU   7          format bad sector
0008            C$READ   EQU   8          read sector(s)
0009            C$WRPROT EQU   9          write protect sector
000A            C$WRITE  EQU   10         write sector(s)
000B            C$SEEK   EQU   11         initiate seek
00C0            C$SELFMT EQU   $C0        select floppy format

        * CSR Register Bit Definitions

0040            SR$RESET EQU   $40        Master Reset
0008            SR$PAR   EQU   $08
0004            SR$INTER EQU   $04        Interrupt Enable
0002            SR$DONE  EQU   $02        Command Done
0001            SR$RDY   EQU   $01        Command Ready

        * CIR Register bit definitions

0001            CIR$PAR  equ   $01        Parity Error
0002            CIR$ERR  equ   $02        Controller/Drive Error

        * Disk format select codes

0000            FMT$SSSD equ   0          sgl side sgl dens
0001            FMT$DSSD equ   1          dbl side sgl dens
0006            FMT$SSDD equ   6          sgl side dbl dens
0007            FMT$DSDD equ   7          dbl side dbl dens
```

```
       ****************************************************
       *
       * DTC 1403 DEVICE DRIVER MODULE HEADER
       *
  00E1              Type     SET   Drivr+Objct
  0081              Revs     SET   Reent+1
  0000 87CD01C4     DSKMOD   MOD   DSKEND,DSKNAM,Type,Revs,DSKENT,DSKSTA
  000D FF                    FCB   %11111111  set all capabilities
  000E 6474E3       DSKNAM   fcs   /dtc/
  0011 04                    FCB   4          EDITION TELLTALE BYTE


       ***************************************************************
       *
       * STATIC STORAGE
       *
D 0055                       ORG   DRVBEG+DRVMEM*2
D 0055              V.ERRBLK rmb   4          error sense block
D 0059              V.STATUS rmb   1          operation status
D 005A              V.CMDBLK EQU   .          command packet begins here
D 005A              V.OPCODE RMB   1          command code byte
D 005B              V.ADDR0  RMB   1          unit/hi sector address
D 005C              V.ADDR1  RMB   1          middle sector address byte
D 005D              V.ADDR2  RMB   1          lo sector address
D 005E              V.COUNT  RMB   1          sector count/interleave factor
D 005F              V.CNTRL  RMB   1          control byte
D 0060              DSKSTA   EQU   .          TOTAL STATIC REQUIREMENT

       * ENTRY POINT BRANCH TABLE FOR FILE MANAGER ACCESS
       *
W 0012 160030       DSKENT   LBRA  INIDSK     INITIALIZE I/O
  0015 1600A9                LBRA  READSC     READ SECTOR
  0018 16009F                LBRA  WRTDSK     WRITE SECTOR
W 001B 16000F                LBRA  GETSTA     GET STATUS
W 001E 160012                LBRA  PUTSTA     PUT STATUS
W 0021 160000                LBRA  TERM       TERMINATE

  0024              TERM     EQU   *
W 0024 7FE300                clr   CSR.REG    deactive controller
  0027 8E0000                ldx   #0         remove from polling list
  002A 103F2A                OS9   F$IRQ

       * Null return for unused entries
  002D              GETSTA   equ   *          get status: not used
  002D 1CFE                  andcc #^CARRY       return no error
  002F 39                    rts



       * Interrupt polling information packet
       * passed to OS-9
       *
  0030 00           INTRPAK  fcb   0          no flip bits
  0031 02                    fcb   SR$DONE    bit to test for IRQ
  0032 0F                    fcb   15         priority = 15



       *******************************************************
       *
       * PUT STATUS CALL
       *
       *
       *
  0033 AE26         PUTSTA   LDX   PD.RGS,Y
  0035 E602                  ldb   R$B,X      GET STAT CALL
```

```
     0037 C103             CMPB  #SS.Reset  RESTORE CALL?
     0039 2773             BEQ   RESTOR     ..YES; DO IT.
     003B C104             CMPB  #SS.WTrk   WRITE TRACK CALL?
     003D 1027015F         LBEQ  WRTTRK     ..YES; DO IT.
     0041 53               COMB             ...NO; ERROR
     0042 C6D0             ldb   #E$UnkSVC  ERROR CODE
     0044 39               RTS
               *************************************************************
               *
               * INITIALIZE THE STORAGE MODULE
               *
               *  INPUT: (U) POINTER TO GLOBAL STORAGE
               *
               *  OUTPUT: (D) MODIFIED
               *          (X) MODIFIED
               *          (Y) UNCHANGED
               *          (U) UNCHANGED
               *
               *
     0045           INIDSK    EQU   *
     0045 6F45                clr   V.WAKE,U    clear wakeup flag
     0047 86FF                lda   #$ff
     0049 A74B                sta   DRVBEG+DD.TOT+2,U init fake media size
     004B A7C831              sta   DRVBEG+DRVMEM+DD.TOT+2,U same for drive 1
     004E 8602                lda   #2
     0050 A7C4                sta   V.NDRV,U    we have 2 drives
     0052 8640                lda   #SR$RESET   perform master reset
   W 0054 B7E300              sta   CSR.REG     store in CSR register
   W 0057 7FE300              clr   CSR.REG     un-reset controller

               * Add device to IRQ polling list
     005A 308DFFD2            leax  INTRPAK,pcr get polling packet address
     005E CCE300              ldd   #CSR.REG    load address of CSR register
     0061 318D0032            leay  INTREQ,pcr  load address of IRQ service ro
     0065 103F2A              os9   F$IRQ       ADD DEVICE TO POLLING TABLE
               * Select FDC format
     0068 86C0                lda   #C$SELFMT   load command code for disk typ
     006A A7C85A              sta   V.OPCODE,u  store command code
     006D 8620                lda   #$20        select floppy for init select
     006F A7C85B              sta   V.ADDR0,u
     0072 6FC85C              clr   V.ADDR1,u
     0075 6FC85D              clr   V.ADDR2,u
     0078 6FC85E              clr   V.COUNT,u
     007B 8606                lda   #FMT$SSDD   init format code
     007D A7C85F              sta   V.CNTRL,u
     0080 30C85A              leax  V.CMDBLK,u  get block addr
   W 0083 BFE304              stx   CMD.DMA     put in DMA addr reg
     0086 8601                lda   #SR$RDY
   W 0088 B7E300              sta   CSR.REG     trigger command
   W 008B B6E300    INIT2     lda   CSR.REG     wait for done
     008E 8402                anda  #SR$DONE
     0090 27F9                beq   INIT2       loop until done
   W 0092 B6E301              lda   CIR.REG     clear done bit
     0095 5F                  CLRB
     0096 39        RETRN1    RTS



               ***********************************************************
               *
               * INTERRUPT REQUEST SERVICE ROUTINE
               *
               *  CALLS WHEN CONTROLLER INTERRUPT OCCURS
               *  CALLS WAKE UP FOR DRIVER
               *
     0097           INTREQ    EQU   *
```

```
      0097 A645                   lda    V.WAKE,u   DEVICE BUSY?

      0099 2710                   beq    RETINT2    ...NO; IGNORE INTRPT
W     009B F6E301                 ldb    CIR.REG    get status; clr IRQ
      009E E7C859                 stb    V.STATUS,u save status
      00A1 6F45                   clr    V.WAKE,U   clear busy flag
      00A3 C601                   ldb    #S$WAKE    send wakeup signal to main tas
      00A5 103F08                 OS9    F$SEND
      00A8 1CFE         RETINT    andcc  #^CARRY       clear carry
      00AA 39                     rts

      00AB 1A01         RETINT2   ORCC   #CARRY     Signal Not Processed
      00AD 39                     RTS

           *****************************************************
           *
           * RESTORE DRIVE TO TRACK 00
           *
           *  INPUT: (Y) POINTER TO PATH DESCRIPTOR
           *         (U) POINTER TO GLOBAL STORAGE
           *
           *  IF ERROR: (B) ERROR CODE & CARRY SET
           *
           *
           *
      00AE              RESTOR    equu   *
      00AE C600                   ldb    #0
      00B0 8E0000                 ldx    #0         use sector zero
      00B3 1700B3                 lbsr   SETUP
      00B6 8601                   lda    #C$RESTOR  load command code
      00B8 2035                   bra    EXECUTE    issue command



           **********************************************
           *
           * WRITE SECTOR COMMAND
           *
           *  INPUT: (B) MSB OF LOGICAL SECTOR NUMBER
           *         (X) LSB'S OF LOGICAL SECTOR NUMBER
           *         (Y) POINTER TO PATH DESCRIPTOR
           *         (U) POINTER TO GLOBAL STORAGE
           *
           *  IF ERROR: (B) ERROR CODE & CARRY SET
           *
      00BA              WRTDSK    equ    *
      00BA 1700AC                 lbsr   SETUP      setup sector address
      00BD 860A                   lda    #C$WRITE   load command code
      00BF 202E                   bra    EXECUTE    execute command



           ****************************************************
           *
           * READ SECTOR COMMAND
           *
           *  INPUT: (B) MSB OF LOGICAL SECTOR NUMBER
           *        (X) LSB'S OF LOGICAL SECTOR NUMBER
           *        (Y) PTR TO PATH DESCRIPTOR
           *        (U) PTR TO GLOBAL STORAGE
           *
           *  OUTPUT: 256 BYTES OF DATA RETURNED IN BUFFER
           *
           *  IF ERROR: CC=SET, B=ERROR CODE
           *
      00C1              READSC    EQU    *
```

```
      00C1 5D                        tstb                    sector zero?
      00C2 2605                      bne    READSC2          skip if not
      00C4 8C0000                    cmpx   #0               test l.s. bytes
      00C7 2707                      beq    READ0            perform special routine if 0

      00C9 17009D       READSC2      lbsr   SETUP            setup command packet
      00CC 8608                      lda    #C$READ          load command code
      00CE 201F                      bra    EXECUTE          execute read command




               ***************************************************
               *
               *  READ LOGICAL SECTOR ZERO
               *
               *
               * copy disk ID area from track 0 to
               * drive table area
      00D0 8DF7         READ0        bsr    READSC2          read the sector
      00D2 2401                      bcc    READ01           continue if no error
      00D4 39                        rts
      00D5 3430         READ01       pshs   X,Y
      00D7 C615                      ldb    #DD.SIZ          number of bytes to copy
      00D9 3049                      leax   DRVBEG,U         load beg addr of table area
      00DB 6D21                      tst    PD.DRV,Y         which drive?
      00DD 2703                      beq    READ02           skip if drive 0
      00DF 308826                    leax   DRVMEM,X         else move to drive 1 table
      00E2 10AE28       READ02       ldy    PD.BUF,Y         get sector buffer address
      00E5 A6A0         READ03       lda    ,y+              get byte from buffer
      00E7 A780                      sta    ,x+              copy to table
      00E9 5A                        decb                    decr count
      00EA 26F9                      bne    READ03           loop til done
      00EC 5F                        clrb                    return no error
      00ED 35B0                      puls   X,Y,PC



               ********************************************************
               *
               * subroutine EXECUTE: perform disk operation with
               * error retry
               *
               *
      00EF 3402         EXECUTE      pshs   a                save command code
      00F1 8600         EXEC0        lda    #C$TREADY
      00F3 8D04                      bsr    COMMAND          check if unit ready
      00F5 25FA                      bcs    EXEC0            loop if not
      00F7 3502                      puls   a                restore command code
               * fall through to COMMAND to perform operation

               ********************************************************
               *
               * Subroutine COMMAND: issue command to controller
               * Pass command packet to controller, sleep until command
               * completed (by interrupt unless booting), then check status.
               *
               * input:  A = command byte
               *         V.CMDBLK assumed to be initialized
               *
               * output: A,Y
               *         B,X corrupted
               *
      00F9              COMMAND      EQU    *
      00F9 AE28                      ldx    PD.BUF,Y         get buffer addr form PD
    W 00FB BFE302                    stx    DATA.DMA         set data addr DMA reg
```

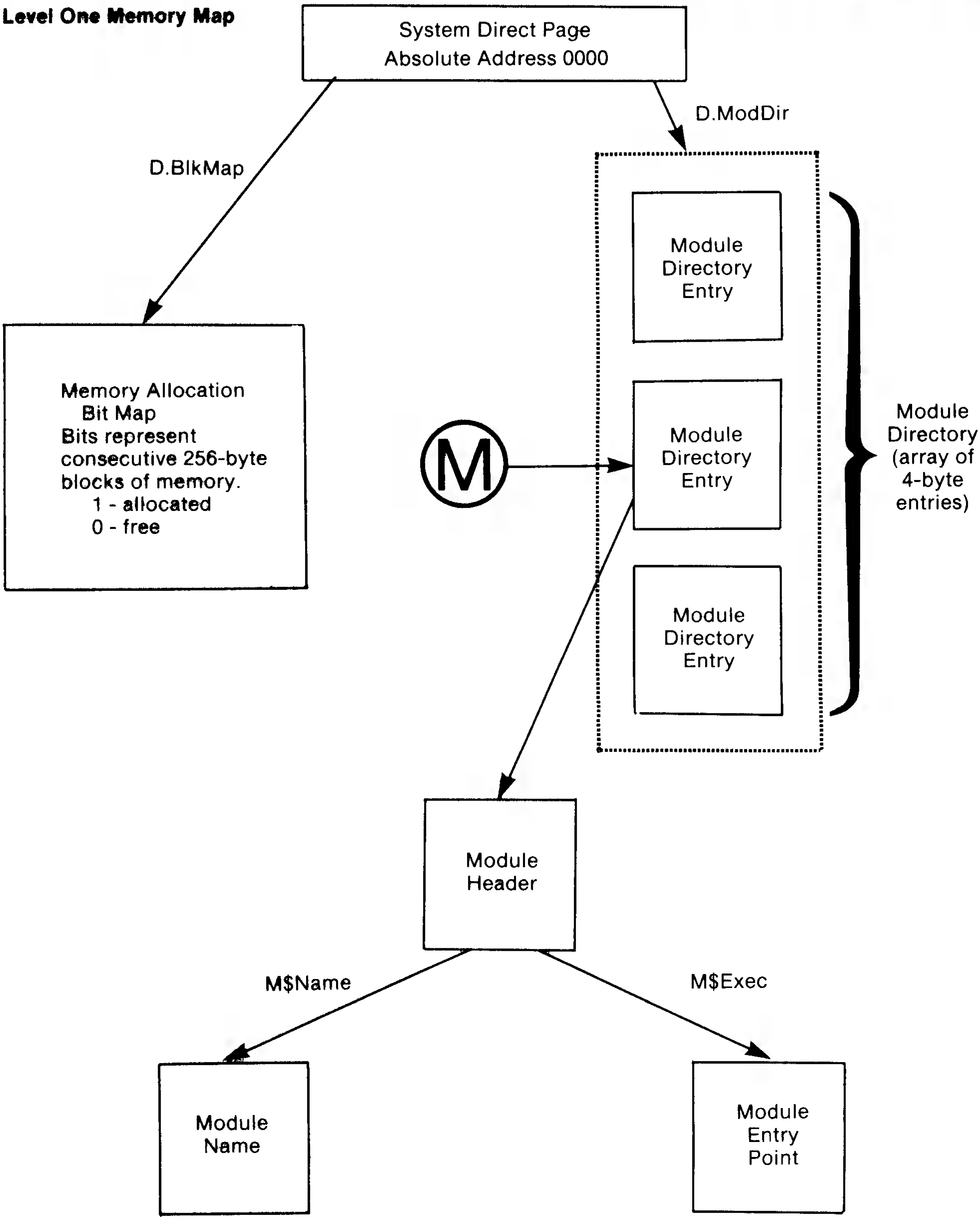
```
  0OFE 30C85A     CMD2     leax  V.CMDBLK,U get addr of cmd block
W 0101 BFE304              stx   CMD.DMA    set command addr DMA reg
  0104 1A50                orcc  #IntMasks  disable IRQ's temporarily
  0106 A7C85A              sta   V.OPCODE,U store command code
  0109 A644                lda   V.BUSY,U   copy busy flag
  010B A745                sta   V.WAKE,U
  010D 8605                lda   #SR$RDY+SR$INTER trigger command w/IRQ
W 010F B7E300              sta   CSR.REG
  0112 1CAF                andcc #^IntMasks enable IRQs now
       * wait for command complete interrupt
  0114 3430                pshs  x,y
  0116 8E0000     CMD6a    ldx   #0         sleep indefinitely
  0119 103F0A              os9   F$Sleep
  011C A645                lda   V.WAKE,u   our wakeup?
  011E 26F6                bne   CMD6a      if not go back to sleep
  0120 3530                puls  x,y        restore regs
       * check for command error
  0122 A6C859              lda   V.STATUS,U read status register
  0125 8402                anda  #CIR$ERR    mask error status
  0127 2603                bne   DECERR     if error, go process
  0129 1CFE                andcc #CARRY     ...else rtn no error
  012B 39                  rts

       * decode error and request controller error detail
  012C 30C855     DECERR   leax  V.ERRBLK,U get sense block addr
W 012F BFE302              stx   DATA.DMA   put in DMA addr reg
  0132 8603                lda   #C$REQDET  request err detail cmd
  0134 8DC8                bsr   CMD2        issue command
  0136 A6C855              lda   V.ERRBLK,U get error byte
  0139 843F                anda  #$3f       mask error type
  013B 1F89                tfr   a,b        copy error code
  013D 308C15              leax  <ERR0TBL,pcr get type 0 error table addr
  0140 C430                andb  #%00110000 mask type code
  0142 270A                beq   LOOKUP     bra if type 0
  0144 308C14              leax  <ERR1TBL,pcr get type 1 error table addr
  0147 C510                bitb  #%00010000 is it type 1?
  0149 2603                bne   LOOKUP     bra if type 1
  014B 308C18              leax  <ERR2TBL,PCR else use type 2 error table
  014E 840F       LOOKUP   anda  #%00001111 mask error code bits
  0150 E686                ldb   a,x        lookup OS-9 error code
  0152 1A01                orcc  #1         set carry bit
  0154 39                  rts              done

       * Error code lookup tables

  0155            ERR0TBL
  0155 F6                  fcb   E$NotRdy
  0156 F6                  fcb   E$NotRdy
  0157 F7                  fcb   E$SEEK
  0158 F6                  fcb   E$NotRdy
  0159 F6                  fcb   E$NotRdy
  015A F7                  fcb   E$SEEK

  015B            ERR1TBL
  015B F4                  fcb   E$Read
  015C F4                  fcb   E$Read
  015D F4                  fcb   E$Read
  015E F4                  fcb   E$Read
  015F F7                  fcb   E$SEEK
  0160 F7                  fcb   E$SEEK
  0161 F7                  fcb   E$SEEK
  0162 F2                  fcb   E$WP
  0163 F4                  fcb   E$Read
  0164 F4                  fcb   E$Read
```

```
0165 F4                     fcb    E$Read

0166            ERR2TBL
0166 FF                     fcb    $FF
0167 F1                     fcb    E$Sect
0168 F9                     fcb    E$BTyp

     **********************************************************
     *
     *  subroutine SETUP: setup command packet
     *
     *  input: B,X = sector number
     *
0169 3414       SETUP       PSHS   B,X          save sector number
016B 8680                   lda    #%10000000 enable retry
016D A7C85F                 sta    V.CNTRL,U
0170 8601                   lda    #1
0172 A7C85E                 sta    V.COUNT,U    set count of one
0175 A621                   lda    PD.DRV,Y     get unit number
0177 4D         Set10       TSTA                is it Winchester?
0178 271B                   BEQ    SetWin       branch if so

     * setup for unit 1 (floppy)
017A 3261                   leas   1,s          MS byte sector # not used
017C 8620                   lda    #%00100000 drive 1 select code
017E A7C85B                 sta    V.ADDR0,u
0181 3506                   puls   d            pop LS bytes sector #
0183 1083000D               cmpd   #13          first track?
0187 2407                   bhs    SETFLD2      bra if not
0189 58                     aslb                multiply sector by 2
018A 49                     rola
018B 6CC85E                 inc    V.COUNT,U    read 2 128 byte sectors
018E 200C                   bra    SETUP4
0190 C3000D     SETFLD2     addd   #13          correct for phantoms
0193 2007                   bra    SETUP4

     * setup for unit 0 (winchester)
0195 3504       SETWIN      puls   b            pop MS sector number
0197 E7C85B                 stb    V.ADDR0,U
019A 3506                   puls   d            pop LS bytes
019C EDC85C     SETUP4      std    V.ADDR1,u
019F 39                     rts

     **********************************************************
     *
     * WRITE TRACK (FOR FORMAT ONLY)
     *
     *
     *
01A0            WRTTRK      EQU    *
01A0 E609                   ldb    R$U+1,x      get track number
01A2 2606                   bne    WRTTRK0      format only track 0
01A4 E607                   ldb    R$Y+1,x      and only side 0
01A6 C501                   bitb   #%00000001
01A8 2701                   beq    WRTTRK1
01AA 39         WRTTRK0     rts
01AB 5F         WRTTRK1     clrb
01AC 8E0000                 ldx    #0           pretend track 00
01AF 8DB8                   bsr    SETUP        init cmd packet
01B1 8603                   lda    #3           interleave for winchester
01B3 6D21                   tst    PD.DRV,Y     which unit?
01B5 2702                   beq    WRTTRK2      skip if winnie
01B7 8603                   lda    #3           else load floppy interleave
01B9 A7C85E     WRTTRK2     sta    V.COUNT,U    put factor in cmd block
01BC 8604                   lda    #C$FORALL    load format command code
01BE 16FF38                 lbra   COMMAND      ...then execute
```

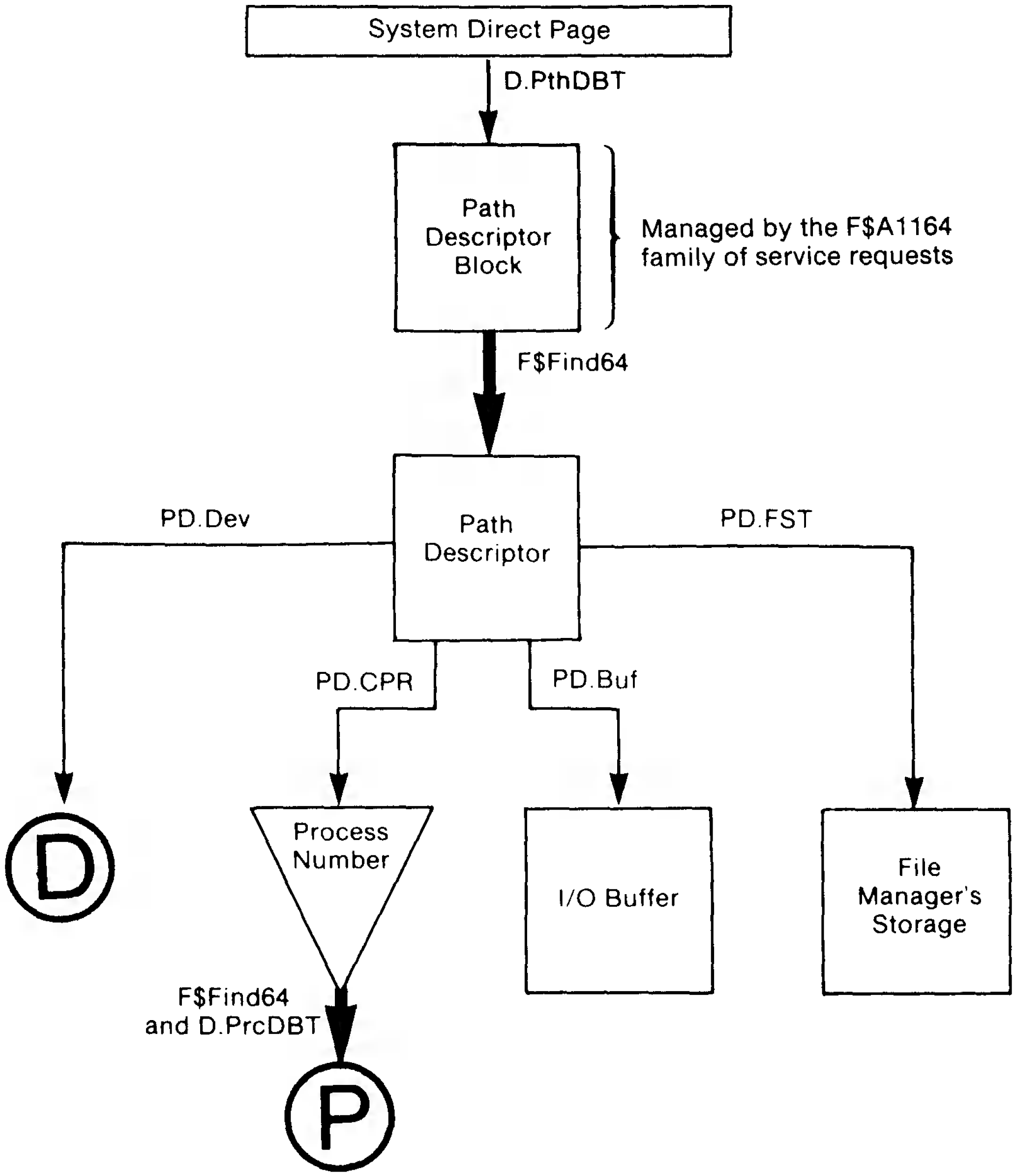
```


01C1 F4837D                EMOD
01C4            DSKEND     EQU   *


                           TTL   Device Descriptors
     ************************************
     *
     * Drive Zero Device Descriptor Module
     *
00F1            Type       SET   DEVIC+OBJCT
0000 87CD002C              MOD   D0End,D0Nam,Type,Revs,D0Mgr,D0Drv
000D FF                    FCB   %11111111  Device Capabilties
000E FF                    FCB   $FF        Controller Port Extended Addre
000F E300                  FDB   ContPort   Controller Port Address
0011 0F                    FCB   D0OptEnd-*-1 Option byte count
0012 0100                  FCB   DT.RBF,0   RBF device, unit 0
0014 008000                FCB   0,$80,0    hard disk, no floppy options
0017 0200                  FDB   512        Number of Cylinders
0019 04                    FCB   4          Tracks per Cylinder
001A 00                    FCB   0          verify on
001B 00200020              FDB   32,32      default sectors/track
001F 03                    FCB   3          sector interleave factor
0020 20                    FCB   32         segment allocation size
0021            D0OptEnd   EQU   *
0021 44B2       D0Nam      FCS   /D2/
0023 5242C6     D0Mgr      FCS   /RBF/      Random Block File Manager
0026 6474E3     D0Drv      FCS   /dtc/      dtc Driver
0029 5655C8                EMOD
002C            D0End      EQU   *



     ************************************
     *
     * Drive One Device Descriptor Module
     *
0000 87CD002C              MOD   D1End,D1Nam,Type,Revs,D1Mgr,D1Drv
000D FF                    FCB   %11111111  Device Capabilities
000E FF                    FCB   $FF        Controller Port Extended Addre
000F E300                  FDB   ContPort   Controller Port Address
0011 0F                    FCB   D1OptEnd-*-1
0012 0101                  FCB   DT.RBF,1   RBF device, unit 1
0014 000101                FCB   0,1,1      slow step,eight inch,double de
0017 004C                  FDB   76         Number of Cylinders
0019 01                    FCB   1          Tracks per Cylinder
001A 00                    FCB   0          verify on
001B 001A000D              FDB   26,13      default sectors/track
001F 03                    FCB   3          sector interleave factor
0020 08                    FCB   8          segment allocation size
0021            D1OptEnd   EQU   *
0021 44B3       D1Nam      FCS   /D3/
0023 5242C6     D1Mgr      FCS   /RBF/      Random Block File Manager
0026 6474E3     D1Drv      FCS   /dtc/      dtc Driver
0029 DD2D5F                EMOD
002C            D1End      EQU   *


                           END

00000 error(s)
00016 warning(s)
$021C 00540 program bytes generated
$000B 00011 data bytes allocated
$1CEA 07402 bytes used for symbols
```

appendix

Level One Memory Map

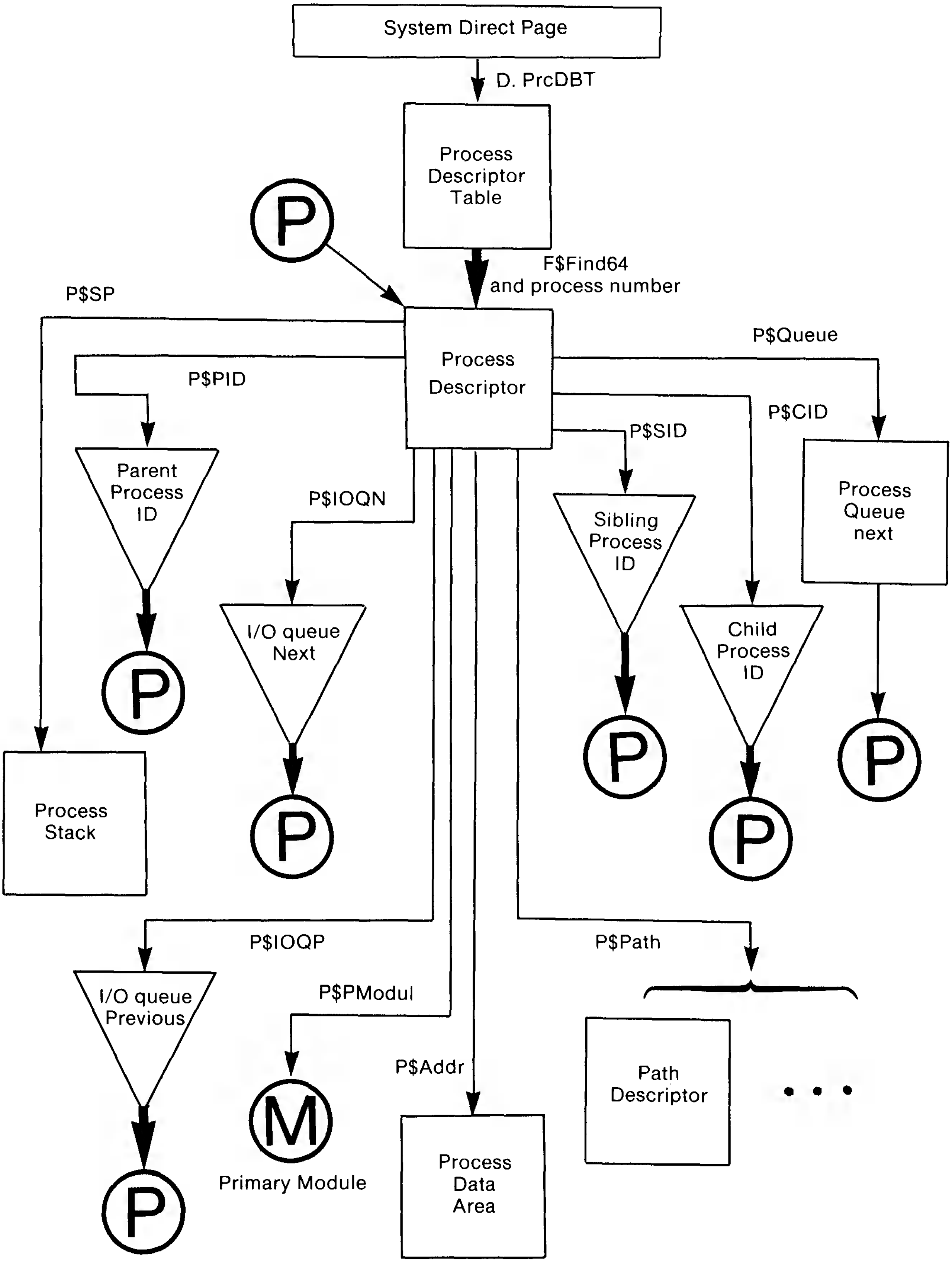


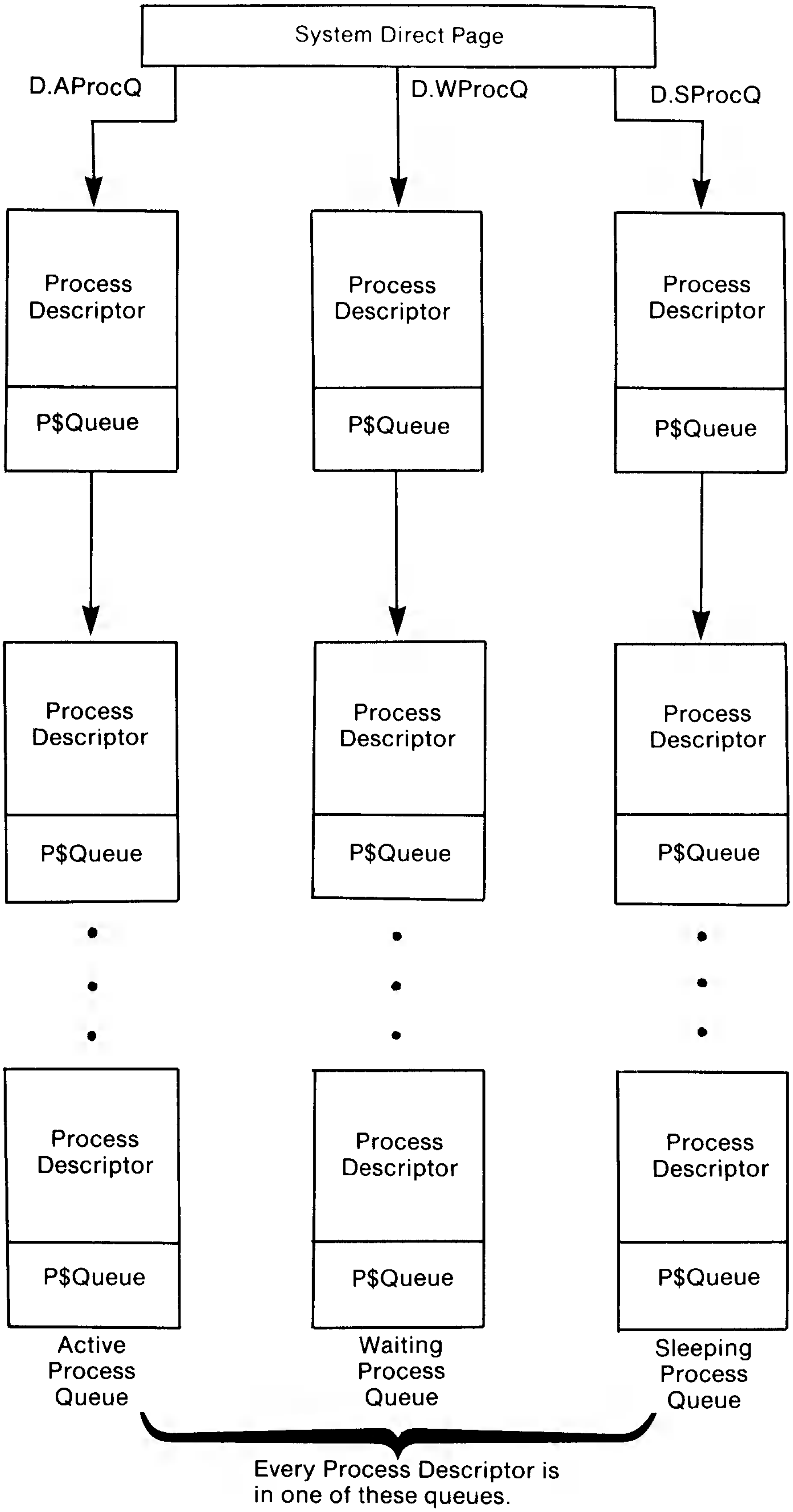
Memory Map Illustration by Eileen O'Malley

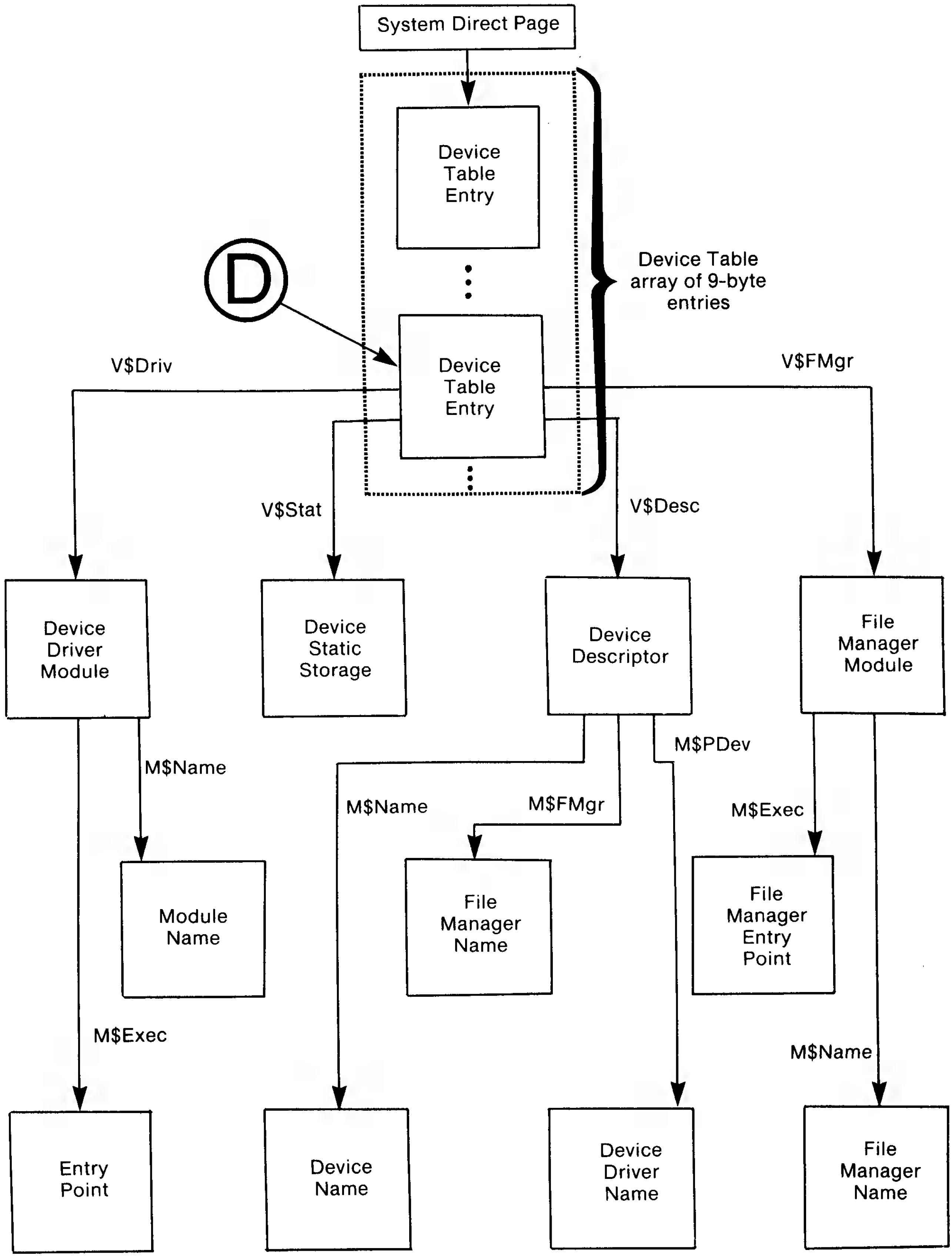


Notes: The large arrow is used to indicate that the actual address of the data area at the end of the arrow isn't to be found in the point structure at the tail of the arrow. To find the structure to which the arrow points, use the F$Find64 SVC.

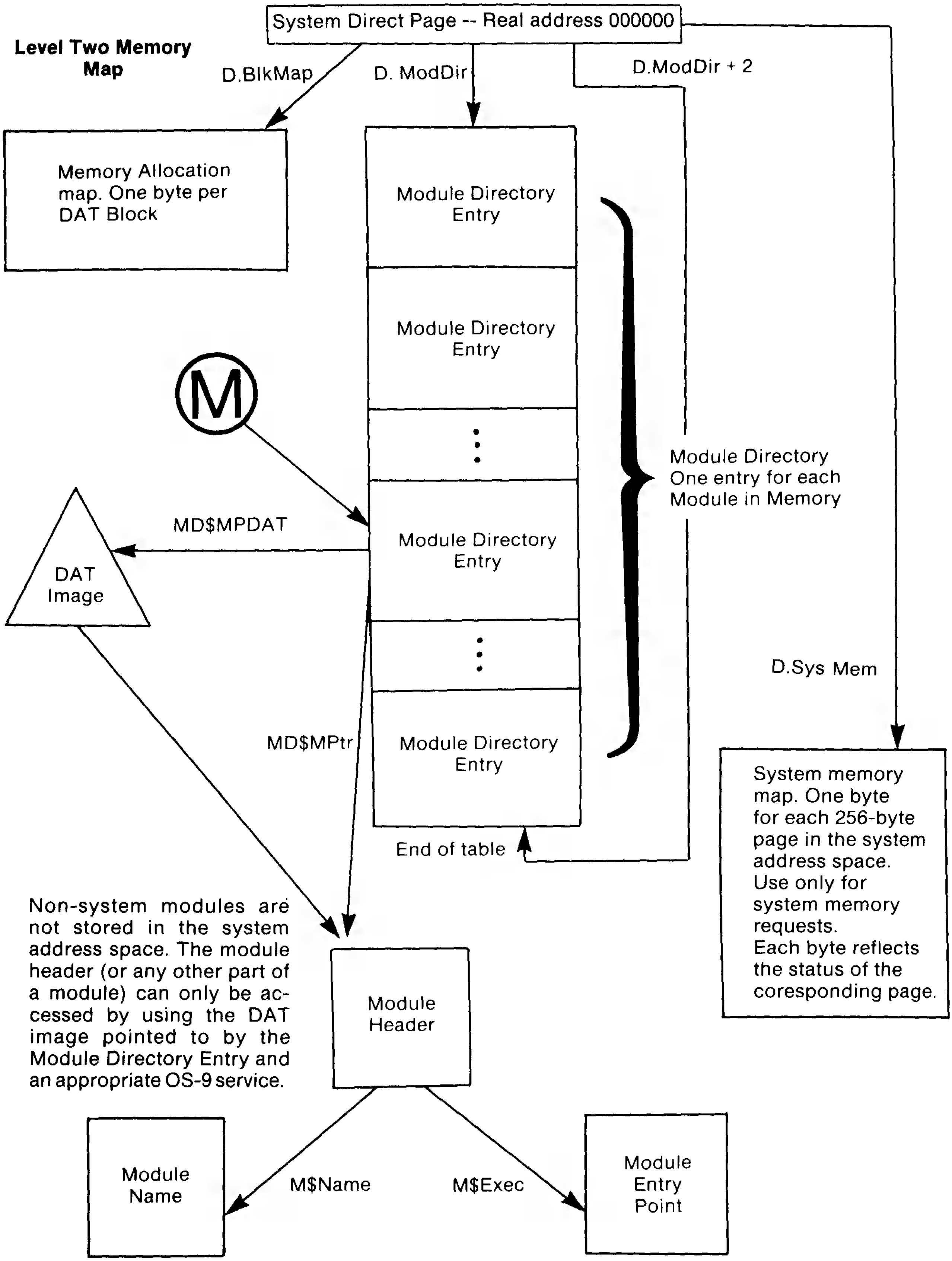
The large triangle containing "process number" is used because the process number isn't a structure. The process number from the path descriptor is used in combination with D.PrcDBT from the system direct page as arguments to F$Find64. The result is a pointer to a process descriptor.


System Direct Page
D. PrcDBT
Process Descriptor Table
F$Find64
and process number
P
P$SP
Process Descriptor
P$Queue
P$PID
P$CID
P$SID
Parent Process ID
P$IOQN
Process Queue next
Sibling Process ID
Child Process ID
I/O queue Next
P
P
P
P
P
Process Stack
P$IOQP
P$Path
P$PModul
I/O queue Previous
P$Addr
Path Descriptor
M
Primary Module
Process Data Area
P

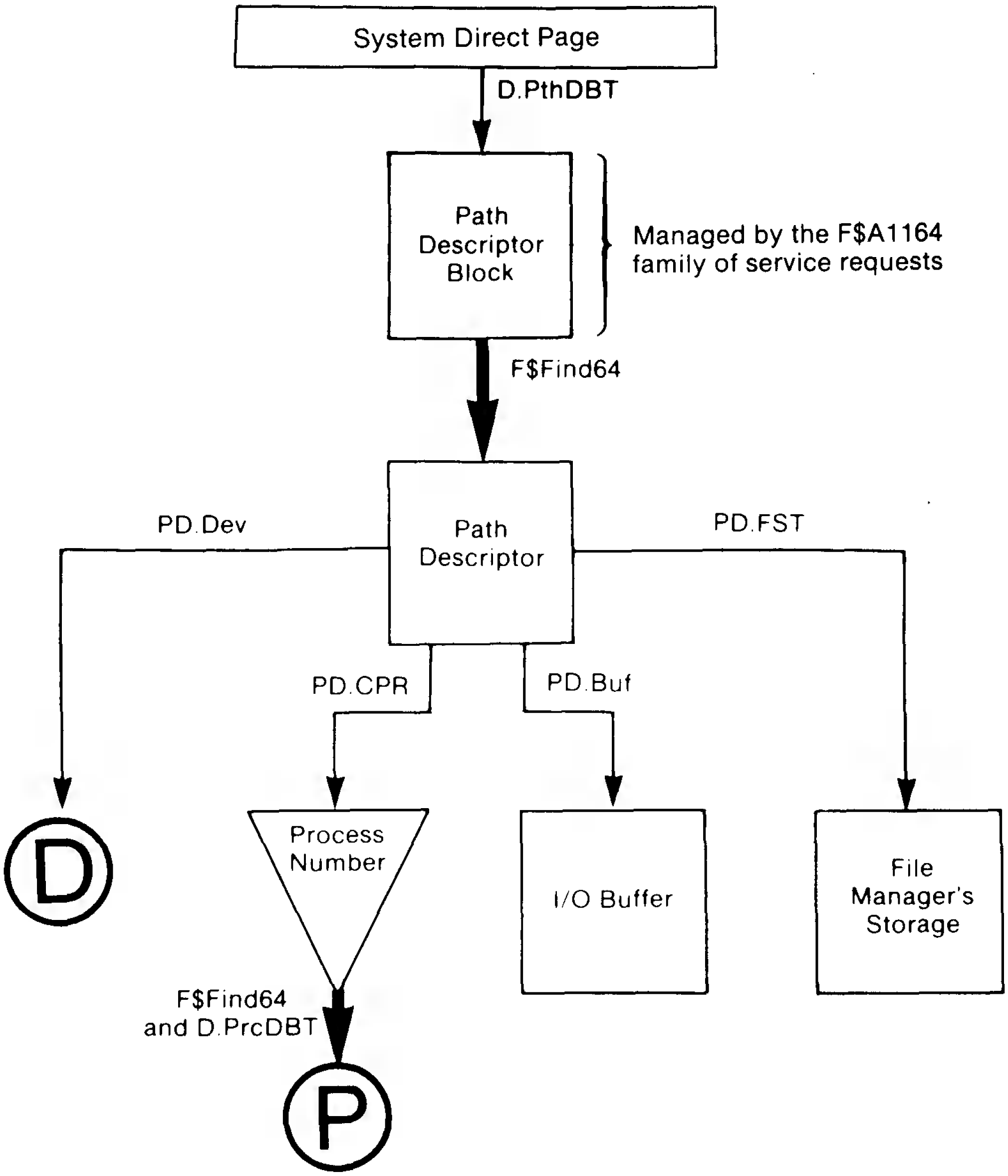

System Direct Page
D.AProcQ
D.WProcQ
D.SProcQ
Process Descriptor
P$Queue
Active Process Queue
Waiting Process Queue
Sleeping Process Queue
Every Process Descriptor is in one of these queues.



System Direct Page
Device Table Entry
D
Device Table Entry
Device Table array of 9-byte entries
V$Driv
V$FMgr
V$Stat
V$Desc
Device Driver Module
Device Static Storage
Device Descriptor
File Manager Module
M$Name
M$PDev
M$Name
M$FMgr
M$Exec
Module Name
File Manager Name
File Manager Entry Point
M$Exec
M$Name
Entry Point
Device Name
Device Driver Name
File Manager Name

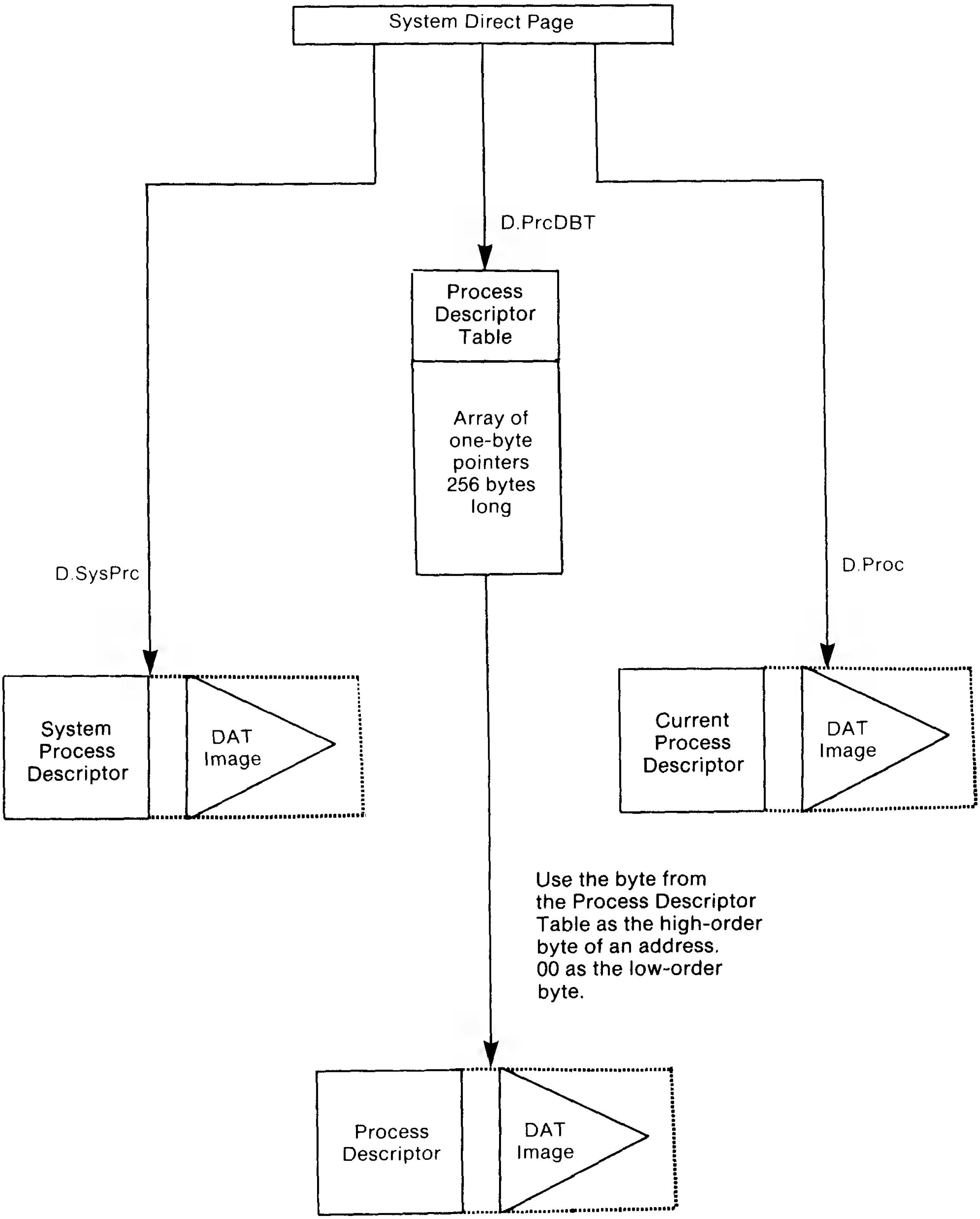


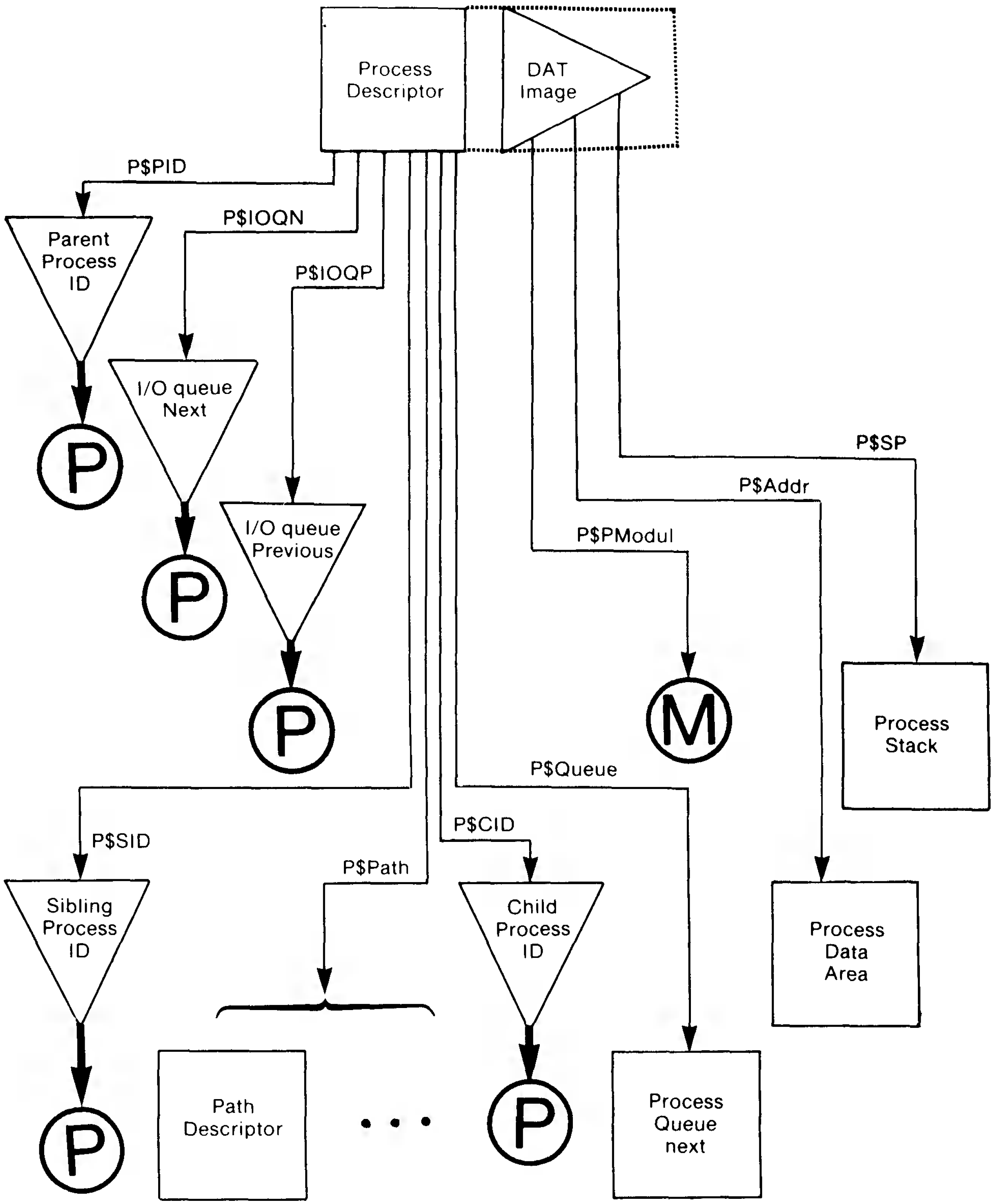
Level Two Memory Map
System Direct Page -- Real address 000000
D.BlkMap
D. ModDir
D.ModDir + 2
Memory Allocation map. One byte per DAT Block
Module Directory Entry
Module Directory Entry
M
Module Directory Entry
Module Directory Entry
Module Directory One entry for each Module in Memory
MD$MPDAT
DAT Image
D.Sys Mem
MD$MPtr
End of table
System memory map. One byte for each 256-byte page in the system address space. Use only for system memory requests. Each byte reflects the status of the coresponding page.
Non-system modules are not stored in the system address space. The module header (or any other part of a module) can only be accessed by using the DAT image pointed to by the Module Directory Entry and an appropriate OS-9 service.
Module Header
Module Name
M$Name
M$Exec
Module Entry Point



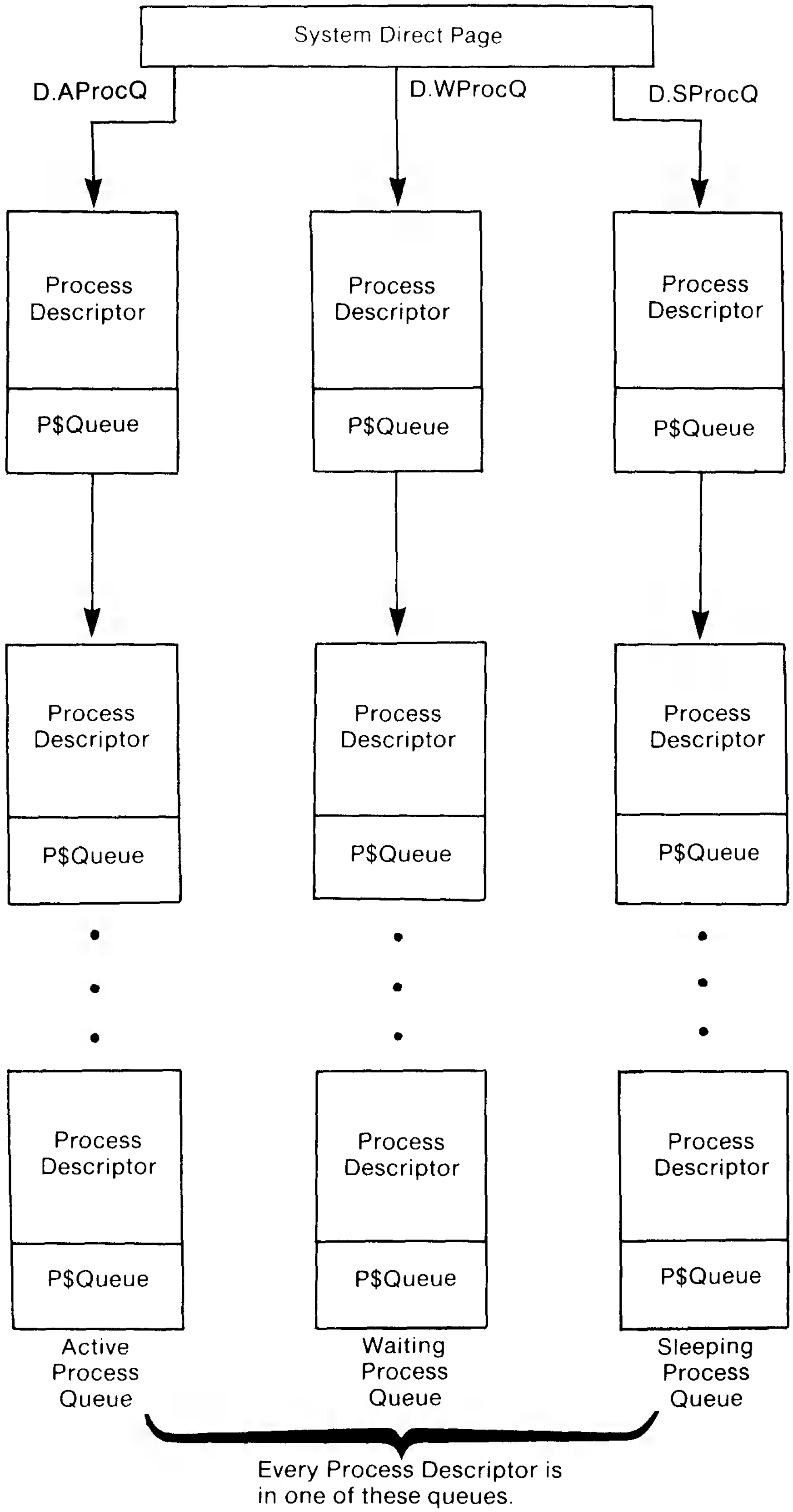
Notes: The large arrow is used to indicate that the actual address of the data area at the end of the arrow isn't to be found in the point structure at the tail of the arrow. To find the structure to which the arrow points, use the F$Find64 SVC.

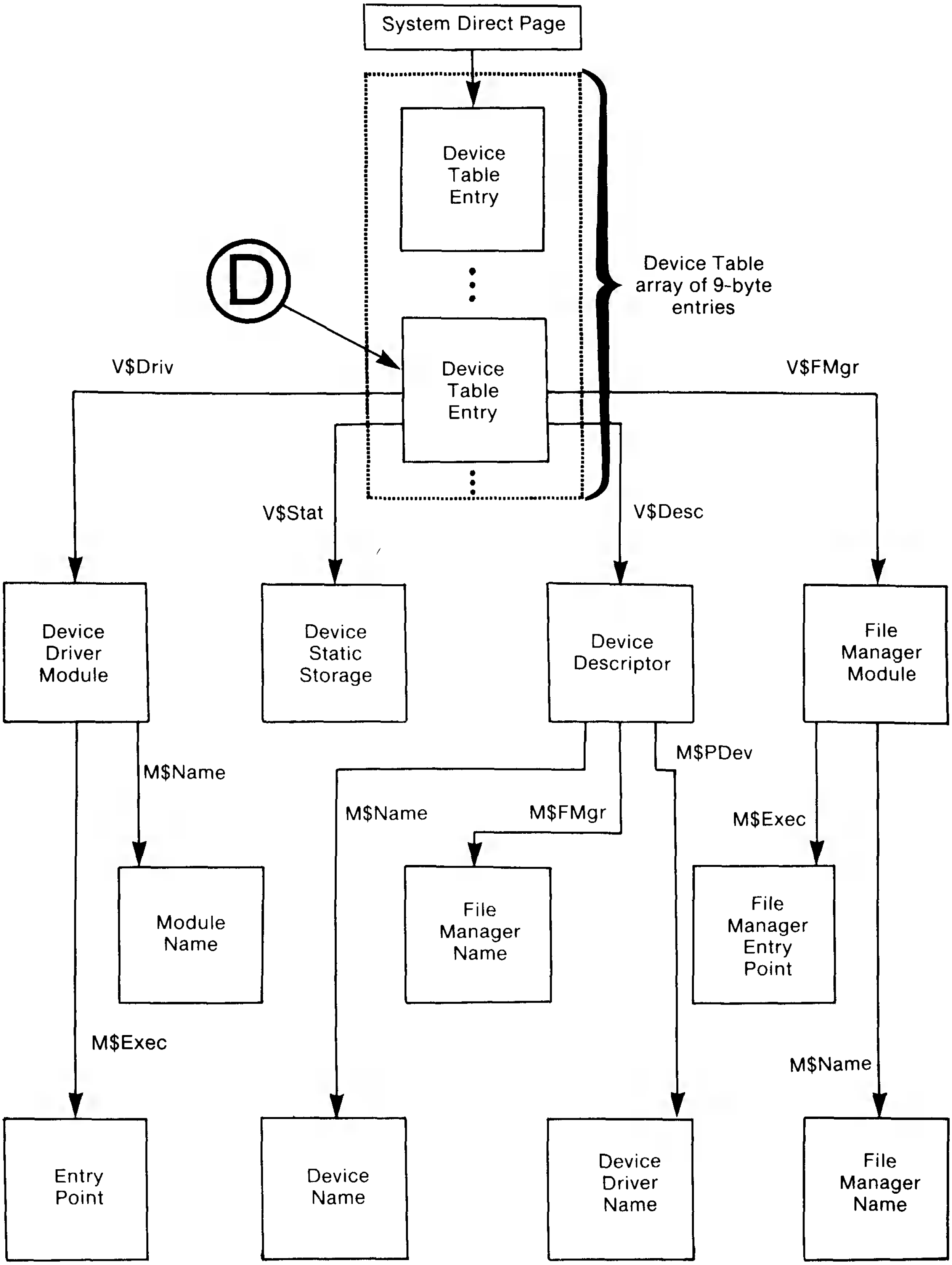
The large triangle containing "process number" is used because the process number isn't a structure. The process number from the path descriptor is used in combination with D.PrcDBT from the system direct page as arguments to F$Find64. The result is a pointer to a process descriptor.


System Direct Page
D.PrcDBT
Process Descriptor Table
Array of one-byte pointers 256 bytes long
D.SysPrc
D.Proc
System Process Descriptor
DAT Image
Current Process Descriptor
DAT Image
Use the byte from the Process Descriptor Table as the high-order byte of an address. 00 as the low-order byte.
Process Descriptor
DAT Image



The pointers PSP, PAddr, and P$Modul in each Process Descriptor are into the process's address space, NOT into the system address space (which is where the Process Descriptor is stored). To use these pointers you must use the DAT image in the Process Descriptor or the process number and an appropriate OS-9 service.


System Direct Page
D.AProcQ
D.WProcQ
D.SProcQ
Process Descriptor
P$Queue
Process Descriptor
P$Queue
Process Descriptor
P$Queue
Process Descriptor
P$Queue
Process Descriptor
P$Queue
Process Descriptor
P$Queue
Process Descriptor
P$Queue
Process Descriptor
P$Queue
Process Descriptor
P$Queue
Active Process Queue
Waiting Process Queue
Sleeping Process Queue
Every Process Descriptor is in one of these queues.


System Direct Page
Device Table Entry
Device Table Entry
Device Table array of 9-byte entries
D
V$Driv
V$FMgr
V$Stat
V$Desc
Device Driver Module
Device Static Storage
Device Descriptor
File Manager Module
M$Name
M$PDev
M$Name
M$FMgr
M$Exec
Module Name
File Manager Name
File Manager Entry Point
M$Exec
M$Name
Entry Point
Device Name
Device Driver Name
File Manager Name

commands and keyword Index

General Index

Falsoft to the Rescue . . .
. . . with The Rainbow Bookshelf

Don't take it out on your local bookseller if you've been frustrated by his woeful supply of books on the Color Computer. There just haven't been enough in circulation.

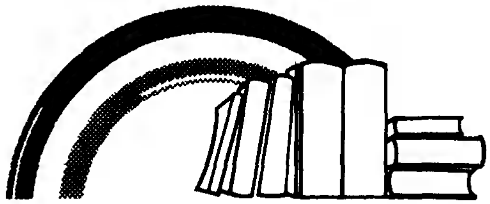
Help is on the way. The same folks who bring you THE RAINBOW are, right now, poring over new manuscripts, considering new concepts, and identifying critical areas of need. There will be fun books, packed with all-new games and informative programs, and books of a more serious nature to help you take full advantage of the Color Computer's capacity.

— **The Rainbow Book of Simulations** is the newest addition to the bookshelf, featuring award winners from THE RAINBOW's very first Simulation contest.
Book **$9.95**
Tape **$9.95**

— **The Complete Rainbow Guide to OS-9**, by Dale Puckett and Peter Dibble. This comprehensive new publication, packed with programs, demystifies the system through a step-by-step process.
Book **$19.95**
Disk (package of 2) **$31.00**

— Copies are still available of **The Rainbow Book of Adventures**, which includes all 13 winners from THE RAINBOW's first Adventure contest. A sure collector's item.
Book **$7.95**
Tape **$7.95**

Order both! The books provide comprehensive instructions often needed to load the programs. The tape/disk saves you hours of time required to key in lengthy listings.



Keep your library up to date. Order now!

I want to start my own Rainbow Bookshelf!

Falsoft, Inc.
The Falsoft Building
9529 U.S. Highway 42
P.O. Box 385
Prospect, KY 40059

Please send me:

☐ The Rainbow Book of Simulations	$ 9.95	______
☐ Rainbow Simulations Tape	$ 9.95	______
☐ The Complete Rainbow Guide to OS-9	$19.95	______
☐ Rainbow Guide to OS-9 Disk (Package of 2)	$31.00	______
☐ The Rainbow Book of Adventures	$ 7.95	______
☐ Rainbow Adventures Tape	$ 7.95	______
Add $1 per book Shipping and Handling in U.S.		______
Canada and Mexico Add $2.00		______
All Other Foreign Add $4.00		______
	Total	______

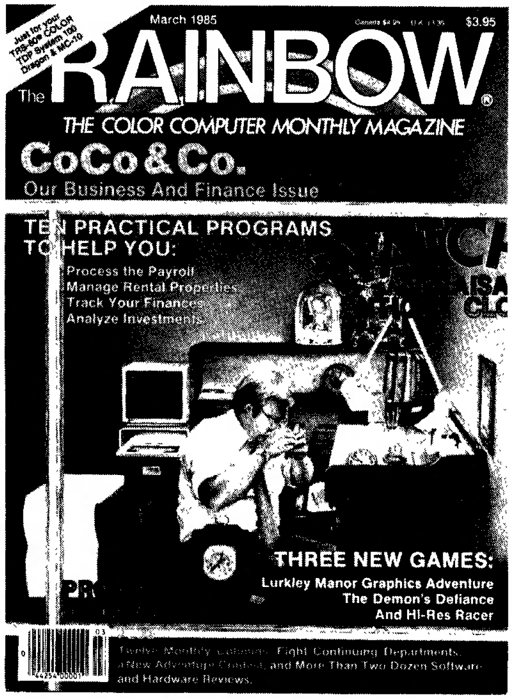
Name ______________________________
Address ______________________________
City ______________ State ______________ ZIP ______
☐ Payment Enclosed ☐ VISA ☐ MasterCard ☐ American Express
Account Number ______________ Interbank No. (MC Only) ______
Signature ______________ Card Expiration Date ______

(Allow 4 weeks for delivery)

Kentucky residents add 5% sales tax.
Sorry — in order to hold down non-editorial cost we do not bill.

The Biggest
The Best
The Indispensable

The Rainbow is the most comprehensive publication a happy CoCo ever had. It's the #1 authority for detailed information on the Color Computer.



The latest news on the Color Computer grapevine is that more and more people are discovering **The Rainbow.**®

Now in its fourth year, **The Rainbow** has become the standard by which other Color Computer magazines are compared. And no wonder! **The Rainbow** towers above the crowd, now offering up to 300 pages each month, including as many as two dozen type-in-and-run program listings, a host of articles and in excess of 30 hardware and software product reviews.

We lead the pack in Color Computer publications and are devoted *exclusively* to the TRS-80® Color, TDP-100 and Dragon-32. We made our climb to the top by continually offering the best and the most by such well-known authors and innovators as Bob Albrecht and Don Inman, and games from top programmers like Robert Tyson, Fred Scerbo and John Fraysse. **The Rainbow** offers the most in entertainment and education, home uses, technical details and hardware projects, tutorials, utilities, graphics and special features like Rainbow Scoreboard and our CoCo Clubs section.

For only $31 a year, you get the keys to all the secrets locked in your CoCo!

Are you searching through the jungle of claims and clamor? Climb above it all. Look up. Find **The Rainbow.**

9529 U.S. Highway 42
The Falsoft Bldg.
P.O. Box 385
Prospect, KY 40059

THE RAINBOW
(502) 228-4492

YES! Sign me up for a year (12 issues) of **THE RAINBOW.**
Name ______
Address ______
City ______ State ______ ZIP ______

Payment Enclosed ☐
Charge Visa ☐ MasterCard ☐ American Express ☐
My Account# ______ Interbank # ______
Signature ______ Card Expiration Date ______

Subscriptions to **the RAINBOW** are $31 a year in the United States. Canadian rate $38 U.S. Surface rate to other countries $68 U.S.; air rate $103 U.S. All subscriptions begin with the current issue. Please allow up to 5-6 weeks for first copy. U.S. FUNDS only. Kentucky residents all 5% sales tax.

Prices subject to change.

Sorry — in order to hold down non-editorial cost we do not bill.

$19.95 ISBN: 0-932471-00-5

The Rainbow Bookshelf

0 44254 16699 01